6.7 Techniques to improve classification accuracy

In this section, you will learn some tricks for increasing classification accuracy. We focus on *ensemble methods*. An ensemble for classification is a composite model, made up of a combination of classifiers. The individual classifiers vote, and a class label prediction is returned by the ensemble based on the collection of votes. Ensembles tend to be more accurate than their component classifiers. We start off in Section 6.7.1 by introducing ensemble methods in general. Bagging (Section 6.7.2), boosting (Section 6.7.3), and random forests (Section 6.7.4) are popular ensemble methods.

Traditional learning models assume that the data classes are well distributed. In many real-world data domains, however, the data are class-imbalanced, where the main class of interest is represented by only a few tuples. This is known as the *class imbalance problem*. We also study techniques for improving the classification accuracy of class-imbalanced data. These are presented in Section 6.7.5.

6.7.1 Introducing ensemble methods

Bagging, *boosting*, and *random forests* are examples of **ensemble methods** (Fig. 6.23). An ensemble combines a series of k learned models (or *base classifiers*), $M_1, M_2, \ldots, M_k$, with the aim of creating an improved composite classification model, $M*$. A given data set, D, is used to create k training sets, $D_1, D_2, \ldots, D_k$, where D_i $(1 \leq i \leq k)$ is used to generate classifier M_i. Given a new data tuple to classify, the base classifiers each vote by returning a class prediction. The ensemble returns a class prediction based on the votes of the base classifiers.

An ensemble tends to be more accurate than its base classifiers. For example, consider an ensemble that performs majority voting. That is, given a tuple $\boldsymbol{X}$ to classify, it collects the class label predictions returned from the base classifiers and outputs the class in the majority. The base classifiers may make mistakes, but the ensemble will misclassify $\boldsymbol{X}$ only if over half of the base classifiers are in error. Ensembles yield better results when there is significant diversity among the models. That is, ideally,

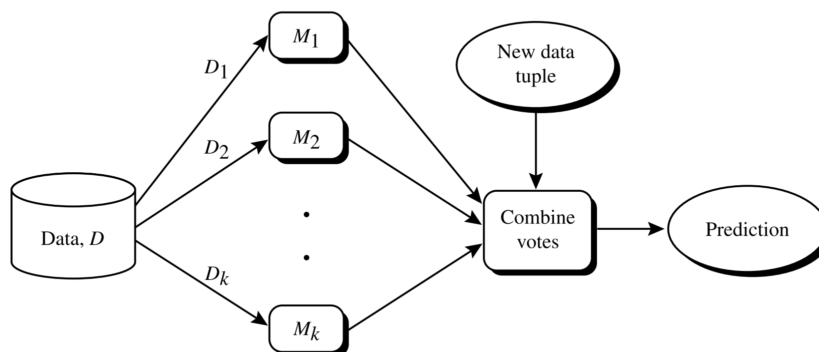


FIGURE 6.23

Increasing classifier accuracy. Ensemble methods generate a set of classification models, $M_1, M_2, \ldots, M_k$. Given a new data tuple to classify, each classifier "votes" for the class label of that tuple. The ensemble combines the votes to return a class prediction.

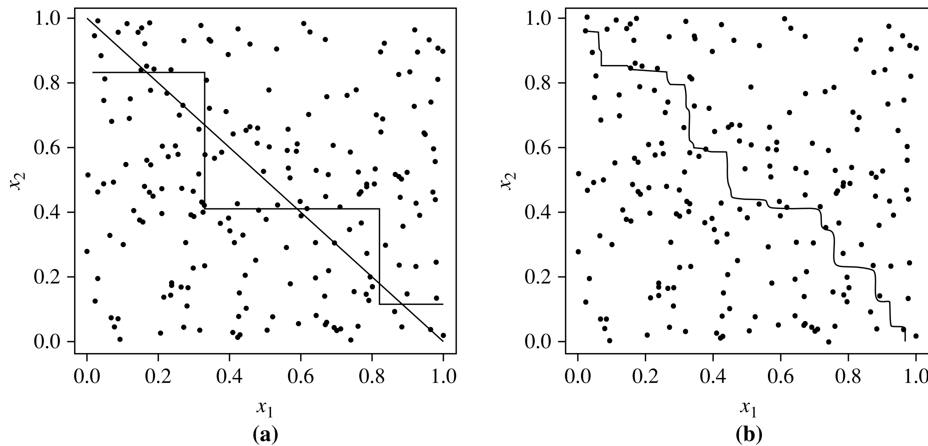


FIGURE 6.24

Decision boundary by (a) a single decision tree and (b) an ensemble of decision trees for a linearly separable problem (i.e., where the actual decision boundary is a straight line). The decision tree struggles with approximating a linear boundary. The decision boundary of the ensemble is closer to the true boundary. *Source:* From Seni and Elder [SE10]. © 2010 Morgan & Claypool Publishers; used with permission.

there is little correlation among classifiers. The base classifiers should also perform better than random guessing. Each base classifier can be allocated to a different CPU and so ensemble methods are parallelizable.

To help illustrate the power of an ensemble, consider a simple two-class problem described by two attributes, x_1 and x_2. The problem has a linear decision boundary. Fig. 6.24(a) shows the decision boundary of a decision tree classifier on the problem. Fig. 6.24(b) shows the decision boundary of an ensemble of decision tree classifiers on the same problem. Although the ensemble's decision boundary is still piecewise constant, it has a finer resolution and is better than that of a single tree.

6.7.2 Bagging

We now take an intuitive look at how bagging works as a method of increasing accuracy. Suppose that you are a patient and would like to have a diagnosis made based on your symptoms. Instead of asking one doctor, you may choose to ask several. If a certain diagnosis occurs more than any other, you may choose this as the final or best diagnosis. That is, the final diagnosis is made based on a majority vote, where each doctor gets an equal vote. Now replace each doctor by a classifier, and you have the basic idea behind bagging. Intuitively, a majority vote made by a large group of doctors may be more reliable than a majority vote made by a small group.

Given a set, D, of d tuples, **bagging** works as follows. For iteration i ($i = 1, 2, \ldots, k$), a training set, D_i, of d tuples is sampled with replacement from the original set of tuples, D. Note that the term *bagging* stands for *bootstrap aggregation*. Each training set is a bootstrap sample, as described in Section 6.6.4. Because sampling with replacement is used, some of the original tuples of D may not

Algorithm: Bagging. The bagging algorithm—create an ensemble of classification models for a learning scheme where each model gives an equally weighted prediction.

Input:

- D, a set of d training tuples;
- k, the number of models in the ensemble;
- a classification learning scheme (e.g., decision tree algorithm, naïve Bayesian, etc.).

Output: The ensemble—a composite model, $M*$.
Method:

(1) **for** $i = 1$ to k **do** // create k models:
(2) create bootstrap sample, D_i, by sampling D with replacement;
(3) use D_i and the learning scheme to derive a model, M_i.
(4) **endfor**

To use the ensemble to classify a tuple, X:

let each of the k models classify $\boldsymbol{X}$ and return the majority vote;

FIGURE 6.25

Bagging.

be included in D_i, whereas others may occur more than once. A classifier model, M_i, is learned for each training set, D_i. To classify an unknown tuple, $\boldsymbol{X}$, each classifier, M_i, returns its class prediction, which counts as one vote. The bagged classifier, $M*$, counts the votes and assigns the class with the most votes to $\boldsymbol{X}$. Bagging can be applied to the prediction of continuous values by taking the average value of each prediction for a given test tuple. The algorithm is summarized in Fig. 6.25.

The bagged classifier often has significantly greater accuracy than a single classifier derived from D, the original training data. It is often more robust to the effects of noisy data and overfitting. The increased accuracy occurs because the composite model reduces the variance of the individual classifiers.

6.7.3 Boosting

We now look at the ensemble method of boosting. As in the previous section, suppose that as a patient, you have certain symptoms. Instead of consulting one doctor, you choose to consult several. Suppose you assign weights to the value or worth of each doctor's diagnosis based on the accuracies of previous diagnoses they have made. The final diagnosis is then a combination of the weighted diagnoses. This is the essence behind boosting.

In **boosting**, weights are also assigned to each training tuple. A series of k classifiers is iteratively learned. After a classifier, M_i, is learned, the weights are updated to allow the subsequent classifier, M_{i+1}, to "pay more attention" to the training tuples that were misclassified by M_i. The final boosted classifier, $M*$, combines the votes of each individual classifier, where the weight of each classifier's vote is a function of its accuracy.

AdaBoost (short for Adaptive Boosting) is a popular boosting algorithm. Suppose we want to boost the accuracy of a learning method. We are given D, a data set of d class-labeled tuples, $(\boldsymbol{X_1}, y_1), (\boldsymbol{X_2}, y_2), \ldots, (\boldsymbol{X_d}, y_d)$, where y_i is the class label of tuple $\boldsymbol{X_i}$. Initially, AdaBoost assigns each training tuple an equal weight of $1/d$. Generating k classifiers for the ensemble requires k rounds

Algorithm: AdaBoost. A boosting algorithm—create an ensemble of classifiers. Each one gives a weighted vote.

Input:

- D, a set of d class-labeled training tuples;
- k, the number of rounds (one classifier is generated per round);
- a classification learning scheme.

Output: A composite model.
Method:

(1) initialize the weight of each tuple in D to $1/d$;
(2) **for** $i = 1$ to k **do** // for each round:
(3) sample D with replacement according to the tuple weights to obtain D_i;
(4) use training set D_i to derive a model, M_i;
(5) compute $error(M_i)$, the error rate of M_i (Eq. (6.34))
(6) **if** $error(M_i) > 0.5$ **then**
(7) abort the loop;
(8) **endif**
(9) **for** each tuple in D that was correctly classified **do**
(10) multiply the weight of the tuple by $error(M_i)/(1 - error(M_i))$; // update weights
(11) normalize the weight of each tuple.
(12) **endfor**

To use the ensemble to classify tuple, X:

(1) initialize weight of each class to 0;
(2) **for** $i = 1$ to k **do** // for each classifier:
(3) $w_i = log\frac{1-error(M_i)}{error(M_i)}$; // weight of the classifier's vote
(4) $c = M_i(X)$; // get class prediction for X from M_i
(5) add w_i to the weight for class c
(6) **endfor**
(7) return the class with the largest weight.

FIGURE 6.26

AdaBoost, a boosting algorithm.

through the rest of the algorithm. We can sample to form any sized training set, not necessarily of size d. Sampling with replacement is used—the same tuple may be selected more than once. Each tuple's chance of being selected is based on its weight. A classifier model, M_i, is derived from the training tuples of D_i. Its error is then calculated using D as the test set. The weights of the tuples are then adjusted according to how they were classified.

If a tuple was incorrectly classified, its weight is increased. If a tuple was correctly classified, its weight is decreased. A tuple's weight reflects how difficult it is to classify—the higher the weight, the more often it has been misclassified. These weights will be used to generate the training samples for the classifier of the next round. The basic idea is that when we build a classifier, we want it to focus more on the misclassified tuples of the previous round. Some classifiers may be better at classifying some "difficult" tuples than others. In this way, we build a series of classifiers that complement each other. The algorithm is summarized in Fig. 6.26.

Now, let's look at some of the math that's involved in the algorithm. To compute the error rate of model M_i, we sum the weights of each of the tuples in D that M_i misclassified. That is,

$$error(M_i) = \sum_{j=1}^{d} w_j \times err(\boldsymbol{X_j}), \tag{6.34}$$

where $err(\boldsymbol{X_j})$ is the misclassification error of tuple $\boldsymbol{X_j}$: If the tuple was misclassified, then $err(\boldsymbol{X_j})$ is 1; otherwise, it is 0. If the performance of classifier M_i is so poor that its error exceeds 0.5, then we abandon it. Instead, we try again by generating a new D_i training set, from which we derive a new M_i.

The error rate of M_i affects how the weights of the training tuples are updated. If a tuple in the round i was correctly classified, its weight is multiplied by $error(M_i)/(1 - error(M_i))$. Once the weights of all the correctly classified tuples are updated, the weights for all tuples (including the misclassified ones) are normalized so that their sum remains the same as it was before. To normalize a weight, we multiply it by the sum of the old weights, divided by the sum of the new weights. As a result, the weights of misclassified tuples are increased, and the weights of correctly classified tuples are decreased, as described before.

"Once boosting is complete, how is the ensemble of classifiers used to predict the class label of a tuple, $\boldsymbol{X}$*?"* Unlike bagging, where each classifier was assigned an equal vote, boosting assigns a weight to each classifier's vote, based on how well the classifier performed. The lower a classifier's error rate, the more accurate it is, and therefore, the higher its weight for voting should be. The weight of classifier M_i's vote is

$$log \frac{1 - error(M_i)}{error(M_i)}. \tag{6.35}$$

For each class, c, we sum the weights of each classifier that assigned class c to $\boldsymbol{X}$. The class with the highest sum is the "winner" and is returned as the class prediction for tuple $\boldsymbol{X}$.

Gradient boosting is another powerful boosting technique, which can be used for classification, regression, and ranking. If we use a tree (e.g., decision tree for classification, regression tree for regression) as the base model (i.e., the weak learner), it is called **gradient tree boosting**, or **gradient boosted tree**. Fig. 6.27 presents the gradient tree boosting algorithm for the regression task. It works as follows.

Gradient tree boosting algorithm starts with a simple regression model $F(x)$ (line 1), which outputs a constant (i.e., the average output of all training tuples). Then, similar to Adaboost, it tries to find a new base model (i.e., weak leaner) $M_t(x)$ at each round (line 3). The newly constructed base model $M_t(x)$ is added to the regression model $F(x)$ (line 8). In other words, the composite regression model $F(x)$ consists of k *additive* base models $M_t(x)$ $(t = 1, ..., k)$. When we search for a new base model $M_t(x)$, all the previously constructed base models (i.e., $M_1(x), ...M_{t-1}(x)$) are kept unchanged.

In order to construct a new base model $M_t(x)$, we first compute the predicted output $\hat{y}_i$ of each training tuple by the current regression model $F(x)$ (line 4) and calculate the **negative gradient** r_i of the loss function with respect to the predicted output $\hat{y}_i$ (line 5). Then, we fit a regression tree model for the training set $\{(x_1, r_1), ..., (x_n, r_n)\}$, where the negative gradient r_i is treated as the targeted output value of the ith training tuple. Since the negative gradient r_i $(i = 1, ..., n)$ changes in different rounds, we end up with different base models $M_t(x)$ $(t = 1, ..., k)$.

"But, why do we use the negative gradients to construct the new base model?" Suppose the loss function $L(y_i, F(x_i)) = \frac{1}{2}(y_i - \hat{y}_i)^2$ (recall that we have used the similar loss function for the regression tree and the least square linear regression model). Then, we can show that the negative gradient

Algorithm: Gradient Tree Boosting for Regression.

Input:

- D, a set of n training tuples $\{(x_1, y_1), ..., (x_n, y_n)\}$, where x_i is the attribute vector of the ith training tuple and y_i is its true target output value;
- k, the number of rounds (one base regression model is generated per round);
- a differential loss function $\text{Loss} = \sum_{i=1}^{n} L(y_i, F(x_i))$.

Output: A composite regression model $F(x)$.
Method:

(1) initialize the regression model $F(x) = \frac{\sum_{i=1}^{n} y_i}{n}$;
(2) **for** $t = 1$ to k **do** // construct a new weak learner $M_t(x)$ for each round:
(3) **for** $i = 1$ to n //each training tuple:
(4) calculate $\hat{y}_i = F(x_i)$; //predicted value by the current model $F(x)$
(5) calculate the negative gradient $r_i = -\frac{\partial L(y_i, \hat{y}_i)}{\partial \hat{y}_i}$;
(6) **endfor**
(7) fit a regression tree model $M_t(x)$ for the training set $\{(x_1, r_1), ..., (x_n, r_n)\}$;
(8) update the composite regression model $F(x) \leftarrow F(x) + M_t(x)$.
(9) **endfor**

FIGURE 6.27

Gradient tree boosting for regression.

$r_i = y_i - \hat{y}_i$, which is the difference between the actual output value and predicted output value by the current regression model $F(x)$ (i.e., the residual). In other words, the negative gradient r_i reveals the "shortcoming" of the current regression model $F(x)$ (i.e., how far away the predicted output is from its actual output value). If we use other loss functions (e.g., the Huber loss in robust regression), the negative gradient is no longer equal to the residual $y_i - \hat{y}_i$, but still provides a good indicator in terms of the prediction quality of the current regression model $F(x)$ on the ith training tuple. For this reason, the negative gradients are also referred to as *pseudo residuals*. By fitting a regression tree model with respect to the negative gradients (i.e., where the "shortcoming" of the current regression model $F(x)$ is), the newly constructed base model, $M_t(x)$, is expected to dramatically improve the composite regression model $F(x)$.

In addition to the algorithm in Fig. 6.27, several alternative design choices for gradient tree boosting exist. For example, similar to Adaboost, we can learn a weight for each base model $M_t(x)$, and then the composite regression model $F(x)$ becomes the *weighted sum* of the k base models. In practice, it was found that shrinking the newly constructed base model helps improve the generalization performance of the composite model $F(x)$ (i.e. $F(x) \leftarrow F(x) + \eta M_t(x)$ in line 8, where $0 < \eta < 1$ is the shrinkage constant.). The number of leaf nodes T of the regression tree $M_t(x)$ plays an important role in the learning performance of the composite model $F(x)$. That is, $F(x)$ might underfit the training set if T is too small, but could overfit the training set with a large T. The typical choice for T is between 4 and 8. At a given round t, we could use a subsample of the entire training set to construct the base model $M_t(x)$. Gradient tree boosting equipped with such a subsampling strategy is referred to as *stochastic gradient (tree) boosting* and it was found to significantly improve the accuracy of the composite model $F(x)$. A highly scalable end-to-end gradient tree boosting system is called **XGBoost**, which is capable to handle a billion-scale training set. XGBoost has made a number of innovations for training gradient

tree boosting, including a new tree construction algorithm designed for sparse data, feature subsampling (as opposed to training tuple subsampling in stochastic gradient boosting), and a highly efficient cache-aware block structure. XGBoost has been successfully used by data scientists in many data mining challenges, often leading to top competitive results.

"How does boosting compare with bagging?" Because of the way boosting focuses on the misclassified tuples, it risks overfitting the resulting composite model to such data. Therefore sometimes the resulting "boosted" model may be less accurate than a single model derived from the same data. Bagging is less susceptible to model overfitting. While both can significantly improve accuracy in comparison to a single model, boosting tends to achieve greater accuracy.

6.7.4 Random forests

We now present another ensemble method called **random forests**. Imagine that each of the classifiers in the ensemble is a *decision tree* classifier so that the collection of classifiers is a "forest." The individual decision trees are generated using a random selection of attributes at each node to determine the split. More formally, each tree depends on the values of a random vector sampled independently and with the same distribution for all trees in the forest. During classification, each tree votes, and the most popular class is returned.

Random forests can be built using bagging (Section 6.7.2) in tandem with random attribute selection. A training set, D, of d tuples is given. The general procedure to generate k decision trees for the ensemble is as follows. For each iteration, i ($i = 1, 2, \ldots, k$), a training set, D_i, of d tuples is sampled with replacement from D. That is, each D_i is a bootstrap sample of D (Section 6.6.4), so that some tuples may occur more than once in D_i, while others may be excluded. Let F be the number of attributes to be used to determine the split at each node, where F is much smaller than the number of available attributes. To construct a decision tree classifier, M_i, randomly select, at each node, F attributes as candidates for the split at the node. The CART methodology is used to grow the trees. The trees are grown to maximum size and are not pruned. Random forests formed this way, with *random input selection*, are called Forest-RI.

Another form of random forest, called Forest-RC, uses *random linear combinations* of the input attributes. Instead of randomly selecting a subset of the attributes, it creates new attributes (or features) that are a linear combination of the existing attributes. That is, an attribute is generated by specifying L, the number of original attributes to be combined. At a given node, L attributes are randomly selected and added together with coefficients that are uniform random numbers on $[-1, 1]$. F linear combinations are generated, and a search is made over these for the best split. This form of random forest is useful when there are only a few attributes available, so as to reduce the correlation between individual classifiers.

Random forests are comparable in accuracy to AdaBoost, yet are more robust to errors and outliers. The generalization error for a forest converges as long as the number of trees in the forest is large. Thus, overfitting is less likely to be a problem. The accuracy of a random forest depends on the strength of the individual classifiers and a measure of the dependence between them. The ideal is to maintain the strength of individual classifiers without increasing their correlation. Random forests are insensitive to the number of attributes selected for consideration at each split. Typically, up to $log_2 d + 1$ are chosen. (An interesting empirical observation was that using a single random input attribute may result in good accuracy that is often higher than when using several attributes.) Because random forests consider much

fewer attributes for each split, they are efficient on very large databases. They can be faster than either bagging or boosting. Random forests give internal estimates of variable importance.

6.7.5 Improving classification accuracy of class-imbalanced data

In this section, we revisit the *class imbalance problem*. In particular, we study approaches to improving the classification accuracy of class-imbalanced data.

Given two-class data, the data are class-imbalanced if the main class of interest (the positive class) is represented by only a few tuples, while the majority of tuples represent the negative class. For multiclass-imbalanced data, the data distribution of each class differs substantially where, again, the main class or classes of interest are rare. The class imbalance problem is closely related to cost-sensitive learning, wherein the costs of errors per class are not equal. In medical diagnosis, for example, it is much more costly to falsely diagnose a cancerous patient as healthy (a false negative) than to misdiagnose a healthy patient as having cancer (a false positive). A false negative error could lead to the loss of life and therefore is much more expensive than a false positive error. Other applications involving class-imbalanced data include fraud detection, the detection of oil spills from satellite radar images, and fault monitoring.

Traditional classification algorithms aim to minimize the number of errors made during classification. They assume that the costs of false positive and false negative errors are equal. By assuming a balanced distribution of classes and equal error costs, they are therefore not suitable for class-imbalanced data. Earlier parts of this chapter presented ways of addressing the class imbalance problem. Although the accuracy measure assumes that the cost of classes are equal, alternative evaluation metrics can be used that consider the different types of classifications. Section 6.6.1, for example, presented *sensitivity* or recall (the true positive rate) and *specificity* (the true negative rate), which help to assess how well a classifier can predict the class label of imbalanced data. Additional relevant measures discussed include F_1 and F_β. Section 6.6.6 showed how ROC curves plot *sensitivity* vs. $1 - $ *specificity* (i.e., the false positive rate). Such curves can provide insight when studying the performance of classifiers on class-imbalanced data.

In this section, we look at general approaches for *improving* the classification accuracy of class-imbalanced data. These approaches include (1) oversampling, (2) undersampling, (3) threshold moving, and (4) ensemble techniques. The first three do not involve any changes to the construction of the classification model. That is, oversampling and undersampling change the distribution of tuples in the training set; threshold moving affects how the model makes decisions when classifying new data. Ensemble methods follow the techniques described in Section 6.7.2 through Section 6.7.4. For ease of explanation, we describe these general approaches with respect to the two-class imbalanced data problem, where the higher-cost classes are rarer than the lower-cost classes.

Both oversampling and undersampling change the training data distribution so that the rare (positive) class is well represented. **Oversampling** works by resampling the positive tuples so that the resulting training set contains an equal number of positive and negative tuples. **Undersampling** works by decreasing the number of negative tuples. It randomly eliminates tuples from the majority (negative) class until there are an equal number of positive and negative tuples.

Example 6.12. Oversampling and undersampling. Suppose the original training set contains 100 positive and 1000 negative tuples. In oversampling, we replicate tuples of the rare class to form a new training set containing 1000 positive tuples and 1000 negative tuples. In undersampling, we randomly

eliminate negative tuples so that the new training set contains 100 positive tuples and 100 negative tuples. □

Several variations to oversampling and undersampling exist. They may vary, for instance, in how tuples are added or eliminated. For example, the SMOTE algorithm uses oversampling where synthetic tuples are added, which are "close to" the given positive tuples in tuple space.

The **threshold-moving** approach to the class imbalance problem does not involve any sampling. It applies to classifiers that, given an input tuple, return a continuous output value (just like in Section 6.6.6, where we discussed how to construct ROC curves). That is, for an input tuple, $\boldsymbol{X}$, such a classifier returns as output a mapping, $f(\boldsymbol{X}) \rightarrow [0, 1]$. Rather than manipulating the training tuples, this method returns a classification decision based on the output values. In the simplest approach, tuples for which $f(\boldsymbol{X}) \geq t$, for some threshold, t, are considered positive, while all other tuples are considered negative. Other approaches may involve manipulating the outputs by weighting. In general, threshold moving moves the threshold, t, so that the rare class tuples are easier to classify (and hence, there is less chance of costly false negative errors). Examples of such classifiers include naïve Bayesian classifiers (Section 6.3) and neural networks (Chapter 10). The threshold-moving method, although not as popular as over- and undersampling, is simple and has shown some success for the two-class-imbalanced data.

Ensemble methods (Section 6.7.2 through Section 6.7.4) have also been applied to the class imbalance problem. The individual classifiers making up the ensemble may include versions of the approaches described here, such as oversampling and threshold moving.

These methods work relatively well for the class imbalance problem on two-class tasks. Threshold-moving and ensemble methods were empirically observed to outperform oversampling and undersampling. Threshold moving works well even on extremely imbalanced data sets. The class imbalance problem on multiclass tasks is much more difficult where oversampling and threshold moving are less effective. Although threshold-moving and ensemble methods show promise, finding a solution for the multiclass imbalance problem remains an area of future work.

6.8 Summary

- **Classification** is a form of data analysis that extracts models describing data classes. A classifier, or classification model, predicts categorical labels (classes). **Numeric prediction** models continuous-valued functions. Classification and numeric prediction are the two major types of prediction problems.
- **Decision tree induction** is a top-down recursive tree induction algorithm, which uses an attribute selection measure to select the attribute tested for each nonleaf node in the tree. **ID3**, **C4.5**, and **CART** are examples of such algorithms using different attribute selection measures. **Tree pruning** algorithms attempt to improve accuracy by removing tree branches reflecting noise in the data.
- **Naïve Bayesian classification** is based on Bayes' theorem of the posterior probability. It assumes class-conditional independence—that the effect of an attribute value on a given class is independent of the values of other attributes.
- **Linear classifiers** compute a linear weighted combination of the input attribute values, based on which it predicts the class label for a given tuple. **Perceptron** and **logistic regression** are two classic examples of linear classifiers.

CHAPTER

7 Classification: advanced methods

In this chapter, you will learn advanced techniques for data classification. We start with **feature selection and engineering** (Section 7.1). Then, we will introduce **Bayesian belief networks** (Section 7.2), which unlike naïve Bayesian classifiers, do not assume class conditional independence. A powerful approach to classification known as support vector machines is presented in Section 7.3. A **support vector machine** transforms training data into a higher dimensional space, where it finds a hyperplane that separates the data by class using essential training tuples called *support vectors*. Section 7.4 describes **rule-based and pattern-based classification**. For the former, our classifier is in the form of a set of IF-THEN rules, whereas the latter explores relationships between attribute–value pairs that occur frequently in data. This methodology builds on research on frequent pattern mining (Chapters 4 and 5). Classification with weak supervision is introduced in Section 7.5. Section 7.6 introduces various techniques for classification on rich data types, such as stream data, sequence data, and graph data. Other related techniques to classification, such as multiclass classification, distance metric learning, interpretability of classification, reinforcement learning, and genetic algorithms are introduced in Section 7.7.

7.1 Feature selection and engineering

For the classification setting introduced in Chapter 6, in order to train a classifier (e.g., naïve Bayes Classifier, k-nearest-neighbor classifier), we assume that there exists a training set with n tuples, each of which is represented by p attributes or features. *"But, where do these p features come from at the first place?"* Let us consider two scenarios. In the first scenario (**Feature Selection**), you might have collected a large number of (say hundreds or thousands or even more) features. However, most of them might be irrelevant with respect to the classification task or redundant with each other. For example, in order to predict whether an online student will drop out before finishing the program, the student ID is an irrelevant feature. In another example of predicting whether a customer will buy a computer, one of the two features, namely yearly income and monthly income, is redundant since one (e.g., yearly income) can be inferred from the other (e.g., monthly income). Including such irrelevant or redundant features during the classifier training process will not help improve the classification accuracy, yet they are likely to make the trained classifier sensitive to noise, leading to degraded generalization performance. *"How can we select a subset of most relevant features from the initial p input features to train a classification model?* This is the main focus of this section.

In the second scenario (**Feature Engineering**), you might wonder *"How can I construct p features so that all of them are critical for the classification task I have?"* or *"Given the initial p features, how can I transform them into another p' attributes so that these transformed features will be more*

Data Mining. https://doi.org/10.1016/B978-0-12-811760-6.00017-5
Copyright © 2023 Elsevier Inc. All rights reserved.

effective for the given classification task?" These are the questions that *feature engineering* tries to answer. For example, in order to predict whether a regional disease outbreak will occur, one might have collected a large number of features from the health surveillance data, including the number of daily positive cases, the number of daily tests, and the number of daily hospitalization. It turns out a powerful indicator (or feature) to predict the disease outbreak is *weekly positive rate*. In this example, the weekly positive rate, which is the ratio of the number of positive cases and the number of tests of a week, can be constructed (or engineered) based on the initial features (e.g., daily positive cases, daily test cases). In practice, feature engineering plays a very important role in the performance of the classification model. Traditionally, feature engineering requires substantial domain knowledge. Some data transformation techniques (e.g., DWT, DFT, and PCA), that were introduced in Chapter 2 can be viewed as feature engineering methods. The deep learning techniques that we will introduce in Chapter 10 provide an automatic way for feature engineering, capable of generating powerful features from the initial input features. The engineered features are often semantically more meaningful with a significant classification performance improvement.

In this section, we will introduce three types of feature selection methods, namely **filter methods**, **wrapper methods**, and **embedded methods**. A filter method selects features based on some goodness measure that is independent of the specific classification model. A wrapper method combines the feature selection and classifier model construction steps together, and it iteratively uses the currently selected feature subset to construct a classification model, which is in turn used to update the selected feature subset. An embedded method simultaneously constructs the classification model and selects the relevant features. In other words, it *embeds* the feature selection step during the classification model construction step. Fig. 7.1 provides a pictorial comparison of these three methods.

Feature selection can be used for both classification and regression. It can also be applied to unsupervised data mining tasks, such as clustering. For both filter and wrapper methods, we will illustrate them with classification tasks. We will mainly use the linear regression task, which was introduced in Section 6.5, to explain the embedded methods.

7.1.1 Filter methods

A **filter method** selects "good" features based on a certain "goodness" measure of the input features. A filter method is independent of the specific classification model and is often used as a preprocessing step of other feature selection methods (e.g., wrapper or embedded methods). The idea is quite straightforward. Suppose we have p initial features and we wish to select k out of p features (where $k < p$). If we have a goodness score for each feature, we can simply select k features with the highest goodness scores.

"So, how shall we measure the goodness of a feature?" Intuitively, we might say that a feature is good if it is highly *correlated* with the class label we want to predict. Suppose there are n training tuples. We wish to measure the correlation between a feature (i.e., attribute) x and the class label y. How can we measure the correlation between the given feature x and the class label y? If the given feature x is a categorical attribute (e.g., job title), a natural choice is χ^2 test, which was introduced in Section 2.2.3. To be specific, a higher χ^2 value indicates a stronger correlation between the given feature x and the class label y. We select k features with the highest χ^2 values.

"*But, what if the given feature x is a continuous attribute (e.g., yearly income)?"* We have two choices. First, we can discretize the continuous attribute x into a categorical attribute (e.g., high,

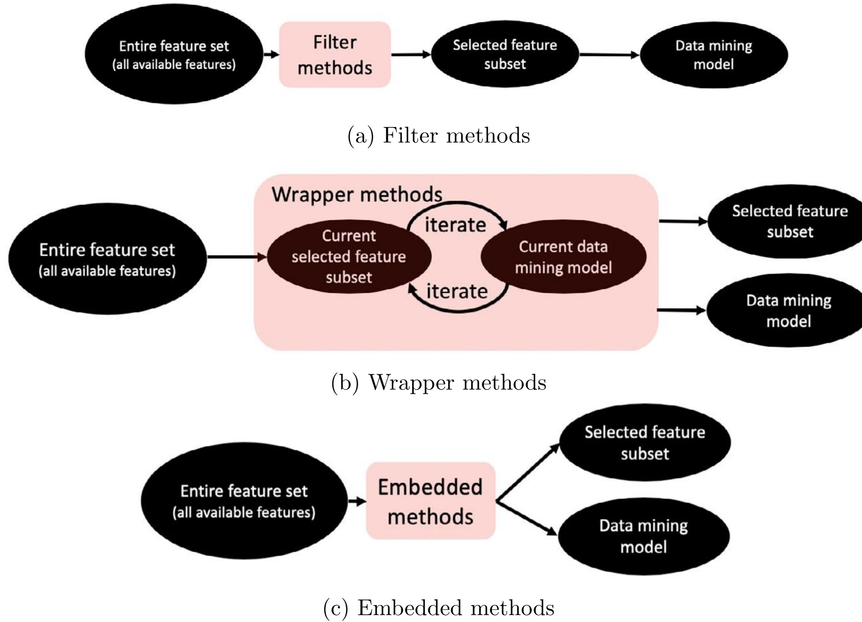


FIGURE 7.1

An overview of three feature selection methods.

medium vs. low income) and then use the χ^2 test to measure the correlation between the discretized attribute and the class label to select k most correlated features. Second, we can resort to **Fisher score** to directly measure the correlation between a continuous variable (the given feature x) and a categorical variable (the class label y).

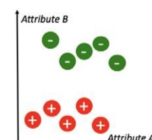
Suppose we have a binary class label y (i.e., whether or not the customer will buy a computer). Intuitively, the feature x (e.g., income) is strongly correlated with the class label y if (1) the average income of all customers who buy a computer is significantly different from the average income of all customers who do not buy a computer, (2) all customers who buy a computer share similar income, and (3) all customers who do not buy a computer share similar income. Formally, Fisher score is defined as follows:

$$s = \frac{\sum_{j=1}^{c} n_j (\mu_j - \mu)^2}{\sum_{j=1}^{c} n_j \sigma_j^2}, \tag{7.1}$$

where c is the total number of classes ($c = 2$ in our example), n_j is the number of training tuples in class j, μ_j and σ_j^2 are the mean value and variance of feature x among all tuples that belong to class j, respectively, and μ is the mean value of feature x among all training tuples. Therefore a feature x would have a high Fisher score if the following conditions hold. First, the *class-specific mean* values $u_j (j = 1, ..., c)$ are dramatically different from each other (e.g., a large numerator of the Fisher score in Eq. (7.1)). Intuitively, this implies that on average, the feature values from different classes are quite different from each other. Second, the *class-specific variance* σ_j^2 is small (e.g., a small denominator of

tuple index	1	2	3	4	5	6	7	8	9	10
Attribute A	1	2	3	4	5	2	3	4	5	6
Attribute B	1.0	0.9	0.8	1.2	1.1	5.1	4.8	4.9	5.2	5.0
Class label	+	+	+	+	+	–	–	–	–	–

(a) Training tuples



(b) Scatter-plot

FIGURE 7.2

Feature section by Fisher score. (a) Ten training tuples, each of which is represented by two attributes (attribute A and attribute B) and a binary class label (+ vs. −). (b) Scatter-plot of training tuples. Intuitively, attribute B better separates the positive training tuples from negative ones than attribute A. This is consistent with Fisher scores: $s(\text{attribute B}) = 200 > s(\text{attribute A}) = 0.125$.

the Fisher score in Eq. (7.1)). This indicates that, within a class, different training tuples share similar feature values.

Example 7.1. We are given 10 training tuples in Fig. 7.2(a), each of which is represented by two attributes (attribute A and attribute B) and a binary class label (+ vs. −). We want to use Fisher scores to decide which attribute is more correlated with the class label. There are five positive tuples and five negative tuples $n_1 = n_2 = 5$. For attribute A, the mean value among all training tuples $\mu = (1+2+3+4+5+2+3+4+5+6)/10 = 3.5$, the mean value among positive training tuples $\mu_1 = (1+2+3+4+5)/5 = 3$, the mean value among negative training tuples $\mu_2 = (2+3+4+5+6)/5 = 4$, the variance of the positive tuples $\sigma_1^2 = ((1-3)^2 + (2-3)^2 + (3-3)^2 + (4-3)^2 + (5-3)^2)/5 = 2$, and the variance of the negative tuples $\sigma_2^2 = ((2-4)^2 + (3-4)^2 + (4-4)^2 + (5-4)^2 + (6-4)^2)/5 = 2$. Therefore the Fisher score for attribute A is $s(\text{attribute A}) = (5 \times (3-3.5)^2 + 5 \times (4-3.5)^2))/(5 \times 2 + 5 \times 2) = 0.125$. We compute the Fisher score for attribute B in a similar way and have that $s(\text{attribute B}) = 200$. According to Fisher scores, attribute B is more correlated with the class label than attribute A. This is consistent with the scatter-plot in Fig. 7.2(b), where the positive tuples are well separated from negative tuples along the vertical axis (attribute B), whereas they are mixed together along the horizontal axis (attribute A). □

In addition to correlation measures (e.g., χ^2 test for categorical feature, Fisher score for continuous feature), we might say that a feature x is good if it contains "a lot of information" about the class label y that we want to predict. This suggests **information-theoretic goodness measures** for feature selection. For example, we can use *information gain* as the goodness measure for feature selection. The information gain, entropy, and conditional entropy were introduced in Section 6.2. In a nutshell, let $H(y)$ be the entropy of the class label y and $H(y|x)$ be the conditional entropy of the class label y given the feature x. The information gain of the feature x is defined as the difference between $H(y)$ and $H(y|x)$. The intuition is that a feature x with a larger information gain can better reduce the impurity (e.g., entropy) of the class label y. Thus it contains "more information" about predicting the class label y. In addition to information gain, another commonly used information theoretic goodness measure for feature selection is *mutual information* (*MI*). Intuitively, the mutual information between a feature x and the class label measures how much information the feature x provides to make the correct prediction of the class label y. Therefore features with the largest mutual information should be selected. The details about how to compute mutual information can be found in Appendix A.

With filter methods, the general process of training a classification model is as follows (Fig. 7.1(a)). Given a set of p initial features, we first use a filter method to select k features (e.g., k out of p features with the highest Fisher scores). Then, using these k selected features, we build a classifier (e.g., a logistic regression classifier). Finally, we evaluate the performance of the trained classifier, such as cross-validation accuracy. Notice that during the feature selection process, a filter method is *independent* of the specific classification model that will be trained with the selected features. Another potential limitation with a filter method is that it does not consider the interaction between different features, and thus might select *redundant* features.

7.1.2 Wrapper methods

A **wrapper method** adopts a different strategy for feature selection by combining the feature selection step and classifier training step together. A wrapper method is often an iterative process (Fig. 7.1(b)). At each iteration, it tries to build a classifier based on the *currently* selected feature subset, and then based on the built classifier, it updates (e.g., add, remove, swap) the selected feature subset. In other words, it *wraps* the feature selection and classifier training together, hence the name of wrapper.

The most important component of a wrapper method is how to search for the best feature subset. A straightforward way (i.e., exhaustive search) is to try all the possible subsets of the p given features. We use each subset of the feature to build a classification model and evaluate its performance, such as the classification accuracy using either the held-out set or cross-validation. The best feature subset is the one with the highest classification accuracy for the given classification model. This strategy is optimal since it finds the best feature subset with the highest classification accuracy. However, it is very expensive in terms of computation, since it needs to search and evaluate all $(2^p - 1)$ possible subsets of the p given features—an exponential number!

In practice, a wrapper method often relies on some heuristic search strategy to avoid the $(2^p - 1)$ exponential search space. Section 2.6.2 introduced different attribute subset selection strategies, which can be applied here. For example, in the *stepwise forward selection* method, it starts with an empty feature subset. At each iteration of the feature selection process, it selects an additional feature, which, when added into the current feature subset, will improve the classification model performance most (e.g., classification accuracy measured by the held-out method or cross-validation). The process will terminate when adding the extra features can no longer improve the classification model performance. In contrast, in the *stepwise backward elimination* method, it starts with all the p initial features, and then iteratively eliminates features from the current subset whose removal would increase the classification accuracy most. We can also combine these two strategies together. That is, at each iteration, we try to select one additional feature and meanwhile might eliminate one existing feature that will improve the classification accuracy most. In addition to these three typical search strategies, some wrapper methods leverage more sophisticated techniques, such as simulated annealing and genetic algorithm. Simulated annealing is a probabilistic optimization technique, often designed for complex (e.g., nonconvex) optimization problems. The latter will be introduced in Section 7.7.

By "wrapping" the feature selection and the classification model construction steps together, a wrapper method tends to have better performance than filtering methods. However, since it needs to iteratively search for the feature subset and (re-)train the classification model, the computational cost of a wrapper method is usually much more intense than filter methods. How can we simultaneously enjoy the advantages of both filter methods and wrapper methods? That is what embedded methods try to answer.

7.1.3 Embedded methods

Embedded methods aim to combine the advantages of both filter methods and wrapper methods. On the one hand, an embedded method performs feature selection and classification model construction simultaneously, so that the two can mutually benefit from each other. On the other hand, an embedded method tries to avoid the expensive, iterative search process in wrapper methods.

We actually have already seen an embedded method in Chapter 6! For decision tree induction that was introduced in Section 6.2, it is possible that only a fraction of all d initial attributes are present in the built decision tree model. For the example in Fig. 6.2, only three attributes (i.e., age, student, credit_rating) are present in the decision tree model, albeit there might be tens of or hundreds of initial attributes. This could happen if the decision tree induction algorithm terminates before it exhausts all d initial attributes, or some attributes of the initially built decision tree are removed during tree pruning process. In either case, all the attributes that appear on the nonleaf tree nodes can be viewed as the selected feature subset, and decision tree induction itself can be viewed as an embedded method for feature selection. In other words, the feature selection process (i.e., to decide which attribute(s) are used as the nonleaf tree nodes) is *embedded* in the decision tree induction process. We simultaneously accomplish both the feature selection step and the classification model construction (i.e., decision tree induction) step. This is the essence of an embedded method.

Other powerful embedded methods often rely on a technique called *sparse learning*. Let us first introduce its high-level idea, and then we will explain the details using linear regression as an example. A handful of data mining models can be solved from the optimization perspective, such as the linear regression model and logistic regression. In a nutshell, we build these data mining models by minimizing some objective (or loss) function that directly or indirectly measures the performance of the corresponding data mining model. For example, in least square linear regression, we find the optimal weight vector w by minimizing the sum of the squared difference between the predicted output and the actual output; in logistic regression, we find the optimal weight vector w by minimizing the negative log likelihood. Now, let us modify the objective function so that it also "penalizes" the number of the features it uses. By minimizing the modified objective function, the trained data mining model might only use a subset of all the d initial features and thus accomplish the task of feature selection. In this way, we will be able to embed the feature selection process (by penalizing the number of features used in the final model) in the model training process.

"So, how can we penalize the number of features used in a data mining model, and how can we solve the modified optimization problem accordingly?" Let us explain the details using least square linear regression (which was introduced in Section 6.5) as an example. Recall that a multilinear regression model assumes $\hat{y}_i = w^T x_i = w_0 + w_1 x_{i,1} + ... + w_d x_{i,d}$, where $\hat{y}_i$ is the predicted output for the ith tuple, $x_i = (1, x_{i,1}, ...x_{i,d})$ is the attribute (feature) vector of the ith tuple, and $w = (w_0, w_1, ..., w_d)$ is the weight vector. We find the optimal weight vector w by minimizing the loss function $L(w) = \frac{1}{2}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 = \frac{1}{2}\sum_{i=1}^{n}(y_i - w^T x_i)^2$, which measures the sum of the squared difference between the predicted output ($\hat{y}_i$) and the actual output (y_i). *"How can we 'embed' the feature selection in the process of training such a linear regression model?"* For the jth feature, if the corresponding weight $w_j = 0$, then it has no contribution on the linear regression model. In other words, this feature is "unselected." This naturally suggests that we can use the l_0 norm[1] of the weight vector w, which

[1] l_0 is a special case of the l_p norm when p approaches 0. l_p norm was introduced in Chapter 2.

counts the number of nonzero elements in the weight vector w, to measure how many features are used (i.e., selected) in the trained linear regression model. Therefore if we train a linear regression model by minimizing the following modified loss function $\tilde{L}(w) = \frac{1}{2}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \lambda\|w\|_0 = \frac{1}{2}\sum_{i=1}^{n}(y_i - w^T x_i)^2 + \lambda\|w\|_0$, the optimal weight vector w is likely to contain some zero elements. Those features whose corresponding weights in vector w are nonzero are selected. The parameter $\lambda > 0$ balances two terms in the modified loss function. Generally speaking, the larger the λ, the less number of the features are likely to be selected (i.e., more elements in the weight vector w are likely to be zeros).

However, finding the optimal weight vector w that minimizes the modified loss function $\tilde{L}(w)$ is very hard. This is because the l_0 norm of the weight vector w, which tells how many features are selected, is *nonconvex*. To address this issue, we replace the l_0 norm by another norm that is convex. It turns out the l_1 norm $\|w\|_1 = \sum_{j=0}^{d}|w_j|$ is the best convex approximation of the l_0 norm, where $|w_j|$ is the absolute value of w_j. Thus we have a new loss function as follows.

$$\hat{L}(w) = \frac{1}{2}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \lambda\|w\|_1 = \frac{1}{2}\sum_{i=1}^{n}(y_i - w^T x_i)^2 + \lambda\sum_{j=0}^{d}|w_j| \tag{7.2}$$

The regression model that minimizes the new loss function $\hat{L}(w)$ in Eq. (7.2) is called **LASSO**, which stands for Least Absolute Shrinkage and Selection Operator. The optimal weight vector w of $\hat{L}$ is often *sparse*, meaning that some of its elements might be zeros. The nonzero elements of the optimal vector w tell that the corresponding features are selected by the linear regression model. Fig. 7.3(a) presents an illustration of the loss function of LASSO (Eq. (7.2)).

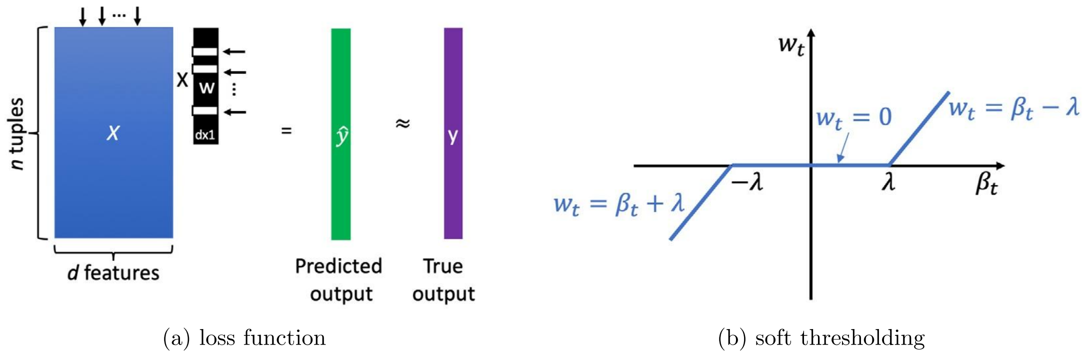


FIGURE 7.3

An illustration of LASSO. (a) Illustration of the loss function of LASSO (Eq. (7.2)). The training set is represented by an $n \times d$ feature matrix X whose rows are tuples and columns are features, and an $n \times 1$ output vector y. By minimizing the sum of the squared difference between the actual and predicted output (i.e., the first term of Eq. (7.2)), the trained linear regression models try to make the predicted output $\hat{y}$ to be as close as possible to the actual output $\hat{y}$. By minimizing the l_1 norm of the weight vector w (i.e., the second term of Eq. (7.2)), some elements of the weight vector w are zeros (indicated by the arrows), and the corresponding features (the columns of the feature matrix X) are "unselected." (b) Soft thresholding pushes the coefficients with small magnitudes (between $-\lambda$ and λ) to be zeros while shrinking the remaining coefficients by λ in magnitude.

"So, how can we find the optimal weight vector w that minimizes $\hat{L}$ in Eq. (7.2)*?"* The good news is that unlike function $\tilde{L}$ that is nonconvex, the loss function $\hat{L}$ in Eq. (7.2) is a convex function. There exist many numerical optimization packages that can be used to solve it. Here, we introduce one of them, called *coordinate descent*.

Unlike the least square regression that has a closed-form solution, the closed-form solution for LASSO does not exist. Coordinate descent finds the optimal weight vector w in an iterative way, and it works as follows. First, we initialize the weight vector w. (We can simply set each element in w as zero.) Then, the algorithm iterates until it converges or some stopping criterion is met, for example, a maximum iteration number has been reached. At each iteration, the algorithm tries to update each element in the weight vector w one-by-one, while fixing all the remaining elements in w. Therefore it boils down to the following question. *"How can we update a single element (say w_t $(0 \leq t \leq d)$) while fixing all other elements?"* We take the following three steps. First, we compute the residual for each training tuple $r_i = y_i - \sum_{j=0, j \neq t}^{d} w_j x_{i,j}$. The intuition of the residual r_i is that it measures the prediction error for the ith tuple if we use all but the tth features. Second, we train a least square regression model for all the input tuples, where each tuple is represented by a single input feature $x_{i,t}$ and its output is the residual r_i. The weight (i.e., coefficient) for the tth feature from such a least square regression model is represented as β_t. (Recall that we can use the closed-form solution of least square regression to find the coefficient β_t, which was introduced in Section 6.5.) The intuition is that if we fix all but the tth features, the coefficient β_t is the best coefficient that minimizes the overall least square prediction error. Third, we update the weight w_t as follows.

$$w_t = \begin{cases} \beta_t - \lambda & \text{if } \beta_t \geq \lambda \\ \beta_t + \lambda & \text{if } \beta_t \leq -\lambda \\ 0 & \text{otherwise} \end{cases} \tag{7.3}$$

The third step is called *soft thresholding*, and its intuition is as follows. If $|\beta_t|$ is greater than the regularization parameter λ, the soft thresholding step would reduce the magnitude of β_t by λ, which is used as the updated coefficient w_t; otherwise the coefficient w_t is simply set as zeros. In other words, the soft thresholding pushes the coefficients with small magnitude as zero while shrinking the remaining coefficients. In this way, the final weight vector w is likely to be sparse with many zero elements, and thus achieves the purpose of feature selection. Fig. 7.3(b) presents an illustration of soft thresholding.

An earlier method for solving LASSO is called **LAR**, which stands for least angle regression. Recall that for linear regression introduced in Section 6.5, we could add the squared l_2 norm of the weight vector into the loss function to prevent overfitting (i.e., Ridge regression). We can add both l_1 norm and the squared l_2 norm into the loss function L. Such a regression model is called **Elastic net**, and the features selected by Elastic net tend to be less correlated with each other, compared with LASSO. We can use a very similar idea as LASSO to embed the feature selection process in the classification model. For example, we can introduce an l_1 norm regularization term in the objective function of logistic regression, so that the weight vector of the trained logistic regression classifier is likely to be sparse. In other words, it only uses a few selected features.

7.2 Bayesian belief networks

Chapter 6 introduced Bayes' theorem and naïve Bayesian classification. In this chapter, we describe *Bayesian belief networks*—probabilistic graphical models, which unlike naïve Bayesian classifiers allow the representation of dependencies among subsets of attributes. Bayesian belief networks can be used for classification. Section 7.2.1 introduces the basic concepts of Bayesian belief networks. In Section 7.2.2, you will learn how to train such models.

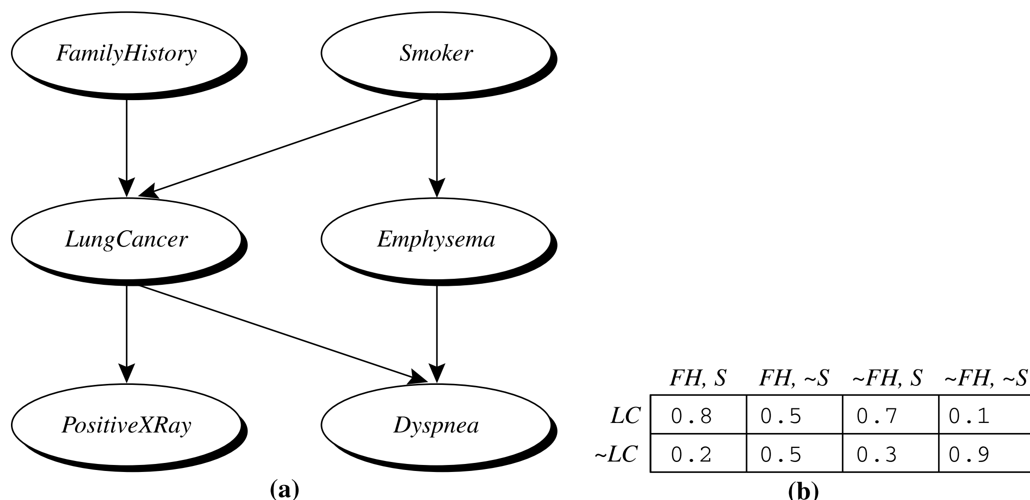
7.2.1 Concepts and mechanisms

The naïve Bayesian classifier makes the assumption of class conditional independence, that is, given the class label of a tuple, the values of the attributes are assumed to be conditionally independent of one another. The benefit of such an assumption is that it significantly simplifies computation. When the assumption holds true, the naïve Bayesian classifier is the most accurate in comparison with all other classifiers. In practice, however, dependencies can exist between variables (i.e., attributes). **Bayesian belief networks** specify joint probability distributions. They allow class conditional independence to be defined between subsets of variables. They provide a graphical model of causal relationships, on which learning can be performed. Trained Bayesian belief networks can be used for classification. Bayesian belief networks are also known as **belief networks**, **Bayesian networks**, and **probabilistic networks**. For brevity, we will refer to them as belief networks.

A belief network is defined by two components—a *directed acyclic graph* and a set of *conditional probability tables* (Fig. 7.4). Each node in the directed acyclic graph represents a random variable. The variables may be discrete- or continuous-valued. They may correspond to actual attributes given in the data or to "hidden variables" believed to form a relationship (e.g., in the case of medical data, a hidden variable may indicate a syndrome, representing a number of symptoms that, together, characterize a specific disease). Each arc represents a probabilistic dependence. If an arc is drawn from a node Y to a node Z, then Y is a **parent** or **immediate predecessor** of Z, and Z is a **descendant** of Y. *Each variable is conditionally independent of its nondescendants in the graph, given its parents.*

Fig. 7.4 is a simple belief network, adapted from Russell, Binder, Koller, and Kanazawa [RBKK95], for six Boolean variables. The arcs in Fig. 7.4(a) allow a representation of causal knowledge. For example, having lung cancer is influenced by a person's family history of lung cancer, as well as whether or not the person is a smoker. Note that the variable *PositiveXRay* is independent of whether the patient has a family history of lung cancer or is a smoker, given that we know the patient has lung cancer. In other words, once we know the outcome of the variable *LungCancer*, then the variables *FamilyHistory* and *Smoker* do not provide any additional information regarding *PositiveXRay*. The arcs also show that the variable *LungCancer* is conditionally independent of *Emphysema*, given its parents, *FamilyHistory* and *Smoker*. On the other hand, we cannot say that *LungCancer* is conditionally independent of *Dyspnea*, given its parents. Why? This is because *Dyspnea* is a child of *LungCancer* in the belief network.

A belief network has one **conditional probability table (CPT)** for each variable. The CPT for a variable Y specifies the conditional distribution $P(Y|Parents(Y))$, where $Parents(Y)$ are the parents of Y. Fig. 7.4(b) shows a CPT for the variable *LungCancer*. The conditional probability for each known value of *LungCancer* is given for each possible combination of the values of its parents. For instance,



	FH, S	FH, ~S	~FH, S	~FH, ~S
LC	0.8	0.5	0.7	0.1
~LC	0.2	0.5	0.3	0.9

FIGURE 7.4

Simple Bayesian belief network. (a) A proposed causal model, represented by a directed acyclic graph. (b) The conditional probability table for the values of the variable *LungCancer* (*LC*) showing each possible combination of the values of its parent nodes, *FamilyHistory* (*FH*) and *Smoker* (*S*). *Source:* Adapted from Russell, Binder, Koller, and Kanazawa [RBKK95].

from the upper leftmost and bottom rightmost entries, respectively, we see that

$$P(LungCancer = yes \,|\, FamilyHistory = yes,\; Smoker = yes) = 0.8$$
$$P(LungCancer = no \,|\, FamilyHistory = no,\; Smoker = no) = 0.9.$$

Let $\boldsymbol{X} = (x_1, \ldots, x_n)$ be a data tuple described by the variables or attributes $Y_1, \ldots, Y_n$, respectively. Recall that each variable is conditionally independent of its nondescendants, given its parents. This allows the belief network to provide a complete representation of the joint probability distribution by the following equation:

$$P(x_1, \ldots, x_n) = \prod_{i=1}^{n} P(x_i | Parents(Y_i)), \tag{7.4}$$

where $P(x_1, \ldots, x_n)$ is the probability of a particular combination of values of $\boldsymbol{X}$, and the values for $P(x_i | Parents(Y_i))$ correspond to the entries in the CPT for attribute Y_i.

A node within the belief network can be selected as an "output" node, representing a class label attribute. There may be more than one output node. Various algorithms for inference and learning can be applied to the network. Rather than returning a single class label, the classification process can return a probability distribution that gives the probability of each class. Belief networks can be used to answer probability of evidence queries (e.g., what is the probability that an individual will have *LungCancer*, given that they have both *PositiveXRay* and *Dyspnea*?) and most probable explanation queries (e.g., which group of the population is most likely to have both *PositiveXRay* and *Dyspnea*?).

Belief networks have been used to model a number of well-known problems. One example is genetic linkage analysis (e.g., the mapping of genes onto a chromosome). By casting the gene linkage problem in terms of inference on Bayesian networks, and using efficient algorithms, the scalability of such analysis has advanced considerably. Other applications that have benefited from the use of belief networks include computer vision (e.g., image restoration and stereo vision), document and text analysis, decision-support systems, financial fraud detection, and sensitivity analysis. The ease with which many applications can be reduced to Bayesian network inference is advantageous in that it curbs the need to invent specialized algorithms for each such application.

7.2.2 Training Bayesian belief networks

"*How does a Bayesian belief network learn?*" In the learning or training of a belief network, a number of scenarios are possible. The network **topology** (or "layout" of nodes and arcs) may be constructed by human experts or inferred from the data. The network variables may be *observable* or *hidden* in all or some of the training tuples. The hidden data case is also referred to as *missing values* or *incomplete data*.

Several algorithms exist for learning the network topology from the training data given observable variables. The problem is one of discrete optimization. For solutions, please see the bibliographic notes at the end of this chapter. Human experts usually have a good grasp of the direct conditional dependencies that hold in the domain under analysis, which helps in network design. Experts must specify conditional probabilities for the nodes that participate in direct dependencies. These probabilities can then be used to compute the remaining probability values.

If the network topology is known and the variables are observable, then training the network is straightforward. It consists of computing the CPT entries, as is similarly done when computing the probabilities involved in naïve Bayesian classification.

When the network topology is given and some of the variables are hidden, there are various methods to choose from for training the belief network. We will describe an effective method based on *gradient descent*, which was also used to train a logistic regression classifier in Chapter 6. For those without an advanced math background, the description of a gradient descent method may look rather intimidating with its calculus-packed formulae. However, packaged software exists to solve these equations. Let us recap the general idea behind a gradient descent method.

Let D be a training set of data tuples, $\boldsymbol{X}_1, \boldsymbol{X}_2, \ldots, \boldsymbol{X}_{|D|}$. Training the belief network means that we must learn the values of the CPT entries. Let w_{ijk} be a CPT entry for the variable $Y_i = y_{ij}$ having the parents $U_i = u_{ik}$, where $w_{ijk} \equiv P(Y_i = y_{ij}|U_i = u_{ik})$. For example, if w_{ijk} is the upper leftmost CPT entry of Fig. 7.4(b), then Y_i is *LungCancer*; y_{ij} is its value, "*yes*"; U_i lists the parent nodes of Y_i, namely, {*FamilyHistory, Smoker*}; and u_{ik} lists the values of the parent nodes, namely, {*"yes," "yes"*}. The w_{ijk} are viewed as weights, analogous to the weights in logistic regression. The set of weights is collectively referred to as $\boldsymbol{W}$. The weights are initialized to random probability values. A *gradient descent* strategy performs greedy hill-descending. At each iteration, the weights are updated and will eventually converge to a local optimum solution.

A **gradient descent** strategy is used to search for the optimal values of certain variables that best minimize an objective function, based on the assumption that each of the possible values is equally likely. Such a strategy is iterative. It searches for a solution along the negative of the gradient (i.e., steepest descent) of an objective function. In our setting, we want to find the set of weights, $\boldsymbol{W}$, that

maximize an objective function.[2] To start with, the weights are initialized to random probability values. The gradient ascent method performs greedy hill-climbing in that, at each iteration or step along the way, the algorithm moves toward what appears to be the best solution at the moment, without backtracking. The weights are updated at each iteration. Eventually, they converge to a local optimum solution.

For our problem, we maximize the objective function $P_w(D) = \prod_{d=1}^{|D|} P_w(\boldsymbol{X}_d)$. This can be done by following the gradient of $\ln P_w(D)$, which makes the problem simpler. (Recall that we have used the same trick to train a logistic regression classifier in Chapter 6.) Given the network topology and initialized w_{ijk}, the algorithm proceeds as follows:

1. **Compute the gradients:** For each i, j, k, compute

$$\frac{\partial \ln P_w(D)}{\partial w_{ijk}} = \sum_{d=1}^{|D|} \frac{\partial \ln(P(Y_i = y_{ij}, U_i = u_{ik} | \boldsymbol{X}_d))}{\partial w_{ijk}}. \tag{7.5}$$

   The probability on the right side of Eq. (7.5) is to be calculated for each training tuple, $\boldsymbol{X}_d$, in D. For brevity, let's refer to this probability simply as p. When the variables represented by Y_i and U_i are hidden for some $\boldsymbol{X}_d$, the corresponding probability p can be computed from the observed variables of the tuple using standard algorithms for Bayesian network inference such as those available in the commercial software package HUGIN (*http://www.hugin.dk*).
2. **Take a small step in the direction of the gradient:** The weights are updated by

$$w_{ijk} \leftarrow w_{ijk} + \eta \frac{\partial \ln P_w(D)}{\partial w_{ijk}}, \tag{7.6}$$

   where η is the **learning rate** representing the step size, and $\frac{\partial \ln P_w(D)}{\partial w_{ijk}}$ is computed from Eq. (7.5). The learning rate is set to a small constant and helps with convergence.
3. **Renormalize the weights:** Because the weights w_{ijk} are probability values, they must be between 0.0 and 1.0, and $\sum_j w_{ijk}$ must equal 1 for all i, k. These criteria are achieved by renormalizing the weights after they have been updated by Eq. (7.6).

Algorithms that follow this learning form are called *adaptive probabilistic networks*. Other methods for training belief networks are referenced in the bibliographic notes at the end of this chapter. Belief networks could be computationally intensive. Because belief networks provide explicit representations of causal structure, a human expert can provide prior knowledge to the training process in the form of network topology or conditional probability values. This can significantly improve the learning speed.

7.3 Support vector machines

In this section, we study **support vector machines (SVMs)**, a method for the classification of both linear and nonlinear data. In a nutshell, an **SVM** is an algorithm that works as follows. It uses a nonlinear

[2] In order to apply gradient descent strategy to maximize, instead of minimize, an objective function, we actually do gradient *ascent* where we update the current solution along the direction of the gradient (i.e., gradient ascent).

mapping to transform the original training data into a higher-dimensional space. Within this new space, it searches for the linear optimal separating hyperplane (i.e., a "decision boundary" separating the tuples of one class from another). With an appropriate nonlinear mapping to a sufficiently high-dimensional space, data from two classes can always be separated by a hyperplane. The SVM finds this hyperplane using *support vectors* ("essential" training tuples) and *margins* (defined by the support vectors). We will delve more into these new concepts later.

"*I've heard that SVMs have attracted a great deal of attention lately. Why?*" The first paper on support vector machines was presented in 1992 by Vladimir Vapnik and colleagues Bernhard Boser and Isabelle Guyon, even though the groundwork for SVMs has been around since the 1960s (including early work by Vapnik and Alexei Chervonenkis on statistical learning theory). Although the training of even the fastest SVMs could be slow, they are highly accurate, owing to their ability to model complex nonlinear decision boundaries. They are much less prone to overfitting than other methods. The support vectors also provide a compact description of the learned model. SVMs can be used for numeric prediction and classification. They have been applied to a number of areas, including handwritten digit recognition, object recognition, emotion recognition, and speaker identification, as well as benchmark time-series prediction tasks.

7.3.1 Linear support vector machines

To explain the mystery of SVMs, let's first look at the simplest case—a two-class problem where the classes are linearly separable. Let the data set D be given as $(\boldsymbol{X}_1, y_1), (\boldsymbol{X}_2, y_2), \ldots, (\boldsymbol{X}_{|D|}, y_{|D|})$, where $\boldsymbol{X}_i$ is the set of training tuples with associated class labels, y_i. Each y_i can take one of two values, either $+1$ or -1 (i.e., $y_i \in \{+1, -1\}$), corresponding to the classes *buys_computer* $=$ *yes* and *buys_computer* $=$ *no*, respectively. To aid in visualization, let's consider an example based on two input attributes, A_1 and A_2, as shown in Fig. 7.5. From the graph, we see that the 2-D data are **linearly separable** (or "linear" for short), because a straight line can be drawn to separate all the tuples of class $+1$ from all the tuples of class -1.

There are an infinite number of separating lines that could be drawn. We want to find the "best" one, that is, one that (we hope) will have the minimum classification error on previously unseen tuples. How can we find this best line? Note that if our data were 3-D (i.e., with three attributes), we would want to find the best separating *plane*. Generalizing to n dimensions, we want to find the best *hyperplane*. We will use "hyperplane" to refer to the decision boundary that we are seeking, regardless of the number of input attributes.

An SVM approaches this problem by searching for the **maximum margin hyperplane**. Consider Fig. 7.6, which shows two possible separating hyperplanes and their associated margins. Before we get into the definition of margins, let's take an intuitive look at this figure. Both hyperplanes can correctly classify all the given data tuples. Intuitively, however, we expect the hyperplane with the larger margin to be more accurate at classifying future data tuples than the hyperplane with the smaller margin. This is why (during the learning or training phase) the SVM searches for the hyperplane with the largest margin, that is, the *maximum marginal hyperplane* (MMH). The associated margin gives the largest separation between classes.

Getting to an informal definition of **margin**, we can say that the shortest distance from a hyperplane to one side of its margin is equal to the shortest distance from the hyperplane to the other side of its margin, where the "sides" of the margin are parallel to the hyperplane. When dealing with the MMH, this distance is, in fact, the shortest distance from the MMH to the closest training tuple of either class.

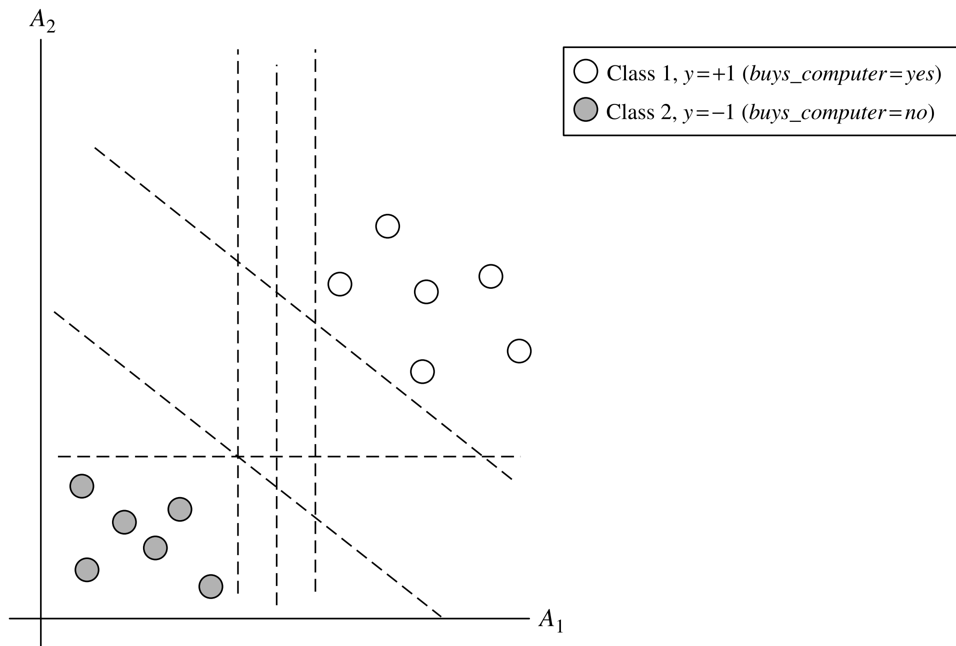


FIGURE 7.5

The 2-D training data that are linearly separable. There are an infinite number of possible separating hyperplanes or "decision boundaries," some of which are shown here as dashed lines. Which one is best?

A separating hyperplane is essentially a linear classifier. Similar to other linear classifiers (such as perceptron, logistic regression) that were introduced in Chapter 7, it can be written as

$$\boldsymbol{W} \cdot \boldsymbol{X} + b = 0, \tag{7.7}$$

where $\boldsymbol{W}$ is a weight vector, namely, $\boldsymbol{W} = \{w_1, w_2, \ldots, w_n\}$; n is the number of attributes; and b is a scalar, often referred to as a bias. To aid in visualization, let's consider two input attributes, A_1 and A_2, as in Fig. 7.6(b). Training tuples are 2-D (e.g., $\boldsymbol{X} = (x_1, x_2)$), where x_1 and x_2 are the values of attributes A_1 and A_2, respectively, for $\boldsymbol{X}$. Eq. (7.7) can be written as

$$b + w_1x_1 + w_2x_2 = 0. \tag{7.8}$$

Thus any point that lies above the separating hyperplane satisfies

$$b + w_1x_1 + w_2x_2 > 0. \tag{7.9}$$

Similarly, any point that lies below the separating hyperplane satisfies

$$b + w_1x_1 + w_2x_2 < 0. \tag{7.10}$$

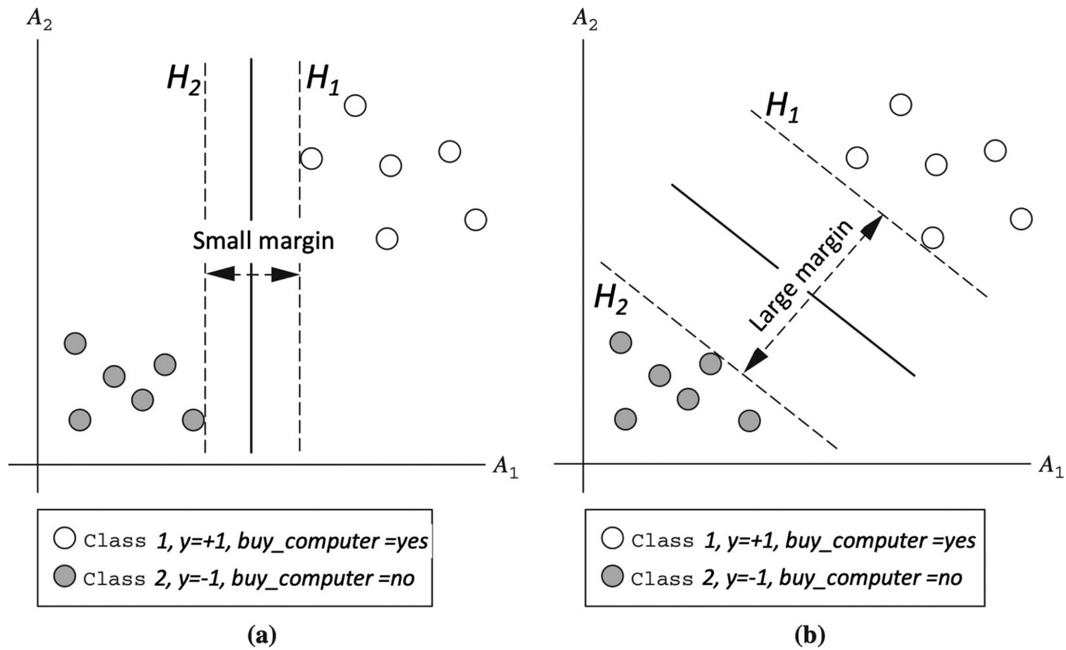


FIGURE 7.6

Here we see just two possible separating hyperplanes and their associated margins. Which one is better? The one with the larger margin (b) should have greater generalization accuracy.

The weights can be adjusted so that the hyperplanes defining the "sides" of the margin can be written as

$$H_1 : b + w_1x_1 + w_2x_2 \geq 1 \quad \text{for } y_i = +1, \tag{7.11}$$

$$H_2 : b + w_1x_1 + w_2x_2 \leq -1 \quad \text{for } y_i = -1. \tag{7.12}$$

That is, any tuple that falls on or above H_1 belongs to class $+1$, and any tuple that falls on or below H_2 belongs to class -1. Combining the two inequalities of Eqs. (7.11) and (7.12), we get

$$y_i(b + w_1x_1 + w_2x_2) \geq 1, \ \forall i. \tag{7.13}$$

Any training tuples that fall on hyperplanes H_1 or H_2 (i.e., $y_i(b + w_1x_1 + w_2x_2) = 1$) are called **support vectors**. That is, they are equally close to the (separating) MMH. In Fig. 7.7, the support vectors are shown encircled with a thicker border. Essentially, the support vectors are the most difficult tuples to classify and give the most information regarding classification.

From this, we can obtain a formula for the size of the maximal margin. The distance from the separating hyperplane to any point on H_1 is $\frac{1}{||\boldsymbol{W}||}$, where $||\boldsymbol{W}||$ is the Euclidean norm of $\boldsymbol{W}$, that is,

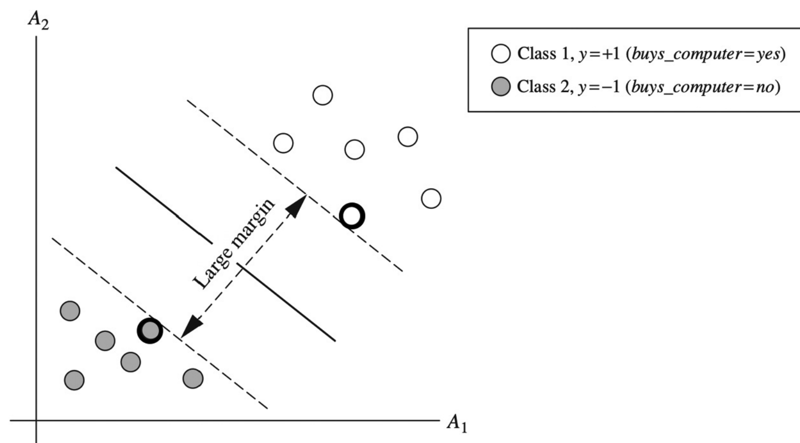


FIGURE 7.7

Support vectors. The SVM finds the maximum margin separating hyperplane, that is, the one with maximum distance between the nearest training tuples. The support vectors are shown with a thicker border.

$\sqrt{\boldsymbol{W} \cdot \boldsymbol{W}}$.[3] By definition, this is equal to the distance from any point on H_2 to the separating hyperplane. Therefore the maximal margin is $\frac{2}{||\boldsymbol{W}||}$. This suggests that we should minimize $\|\mathbf{W}\|^2$ in order to make the margin as large as possible. Notice that if the tuples are in n dimensional space, Eq. (7.13) becomes $y_i(\mathbf{W}'\mathbf{X}_i + b)) \geq 1$. Putting it together, we have the following mathematical formulation of SVM:

$$\begin{aligned} &\min \|\mathbf{W}\|^2, \\ &\text{s.t.} \quad y_i(\mathbf{W}'\mathbf{X}_i + b) \geq 1, \quad \forall i. \end{aligned} \tag{7.14}$$

The intuition of the above formulation is that we want to find a linear classifier (i.e., hyperplane), such that (1) its margin is as large as possible (i.e., $\min \|\mathbf{W}\|^2$), and (2) each training tuple is correctly classified (i.e., $y_i(\mathbf{W}'\mathbf{X}_i + b)) \geq 1, \ \forall i$). The corresponding classifier is often called *hard-margin linear SVM*.

"*So, how does an SVM find the MMH and the support vectors?*" Using some "fancy math tricks," we can rewrite Eq. (7.14) so that it becomes what is known as a (convex) quadratic programming problem. Such fancy math tricks are beyond the scope of this book. Advanced readers may be interested to note that the tricks involve rewriting Eq. (7.14) as its dual form using Lagrangian formulation and then solving for the solution using Karush-Kuhn-Tucker (KKT) conditions. Details can be found in the bibliographic notes at the end of this chapter (Section 7.10).

If the data are relatively small (say, with a few thousand training tuples), any optimization software package for solving convex quadratic programming problems can then be used to find the support vectors and MMH. For larger data, special and more efficient algorithms for training SVMs can be

[3] If $\boldsymbol{W} = \{w_1, w_2, \ldots, w_n\}$, then $\sqrt{\boldsymbol{W} \cdot \boldsymbol{W}} = \sqrt{w_1^2 + w_2^2 + \cdots + w_n^2}$.

used instead, the details of which exceed the scope of this book. Once we've found the support vectors and MMH (note that the support vectors define the MMH!), we have a trained support vector machine. The MMH is a linear class boundary, and so the corresponding SVM can be used to classify linearly separable data.

"*Once I've got a trained support vector machine, how do I use it to classify test (i.e., new) tuples?*" Based on the Lagrangian formulation mentioned before, the MMH can be rewritten as the decision boundary

$$d(\boldsymbol{X}) = \sum_{i=1}^{l} y_i \alpha_i \boldsymbol{X}' \boldsymbol{X}_i + b, \tag{7.15}$$

where y_i is the class label of support vector $\boldsymbol{X}_i$; $\boldsymbol{X}$ is a test tuple and $'$ denotes the transpose of a vector; α_i and b are numeric parameters that were determined automatically by the optimization or SVM algorithm noted before; and l is the number of support vectors, which is often much smaller than the total number of training tuples. Interested readers may note that the α_i are Lagrangian multipliers. For linearly separable data, the support vectors are a subset of the actual training tuples. Slight twist regarding this when dealing with nonlinearly separable data, as we shall see in the following.

Given a test tuple, $\boldsymbol{X}$, we plug it into Eq. (7.15), and then check to see the sign of the result. This tells us on which side of the hyperplane the test tuple falls. If the sign is positive, then $\boldsymbol{X}$ falls above the MMH, and so the SVM predicts that $\boldsymbol{X}$ belongs to class $+1$ (representing *buys_computer* = *yes*, in our case). If the sign is negative, then $\boldsymbol{X}$ falls below the MMH and the class prediction is -1 (representing *buys_computer* = *no*).

Notice that the Lagrangian formulation of our problem Eq. (7.15) contains a dot product between support vector $\boldsymbol{X}_i$ and test tuple $\boldsymbol{X}$. This will prove very useful for finding the MMH and support vectors of a nonlinear SVM when the given data are linearly inseparable, as described further in the next section. However, before that, let's briefly introduce how we can modify the formulation of hard-margin linear SVM (Eq. (7.14)) for the nonlinear case. That is, we still wish to find a linear classifier (i.e. a hyperplane) when the training tuples are linearly inseparable. Here, the trick is that we allow some training tuples to be mis-classified. To be specific, we can introduce a nonnegative slack variable $\xi_i \geq 0$ for each training tuple, $\mathbf{X}_i$. If $\xi_i = 0$, it means that the corresponding tuple $\mathbf{X}_i$ is correctly classified by the hyperplane (i.e., $y_i(\mathbf{W}'\mathbf{X}_i + b) \geq 1$). In other words, a training example with $\xi_i = 0$ is just like the one in the hard-margin linear SVM. However, if $\xi_i > 0$, it means that the tuple $\mathbf{X}_i$ is incorrectly classified by the hyperplane and its magnitude $|\xi|$ indicates how far the training tuple is away from its corresponding side (i.e., H_1 for a positive training example, and H_2 for a negative training example). See Fig. 7.8(a) for an illustration.

Then we have the following alternative mathematical formulation of SVM. The corresponding classifier is often called *soft-margin linear SVM*. Different from hard-margin linear SVM, our new objective function has two terms, including (1) $\|\mathbf{W}\|^2$, which measures the size of margin (i.e., the smaller $\|\mathbf{W}\|^2$, the larger margin), and (2) the sum of all slack variables $\sum_{i=1}^{N} \xi_i$, which measures the (approximate) number of incorrectly classified training tuples (i.e., the training error). In Eq. (7.16), N is the total number of training tuples and $C > 0$ is a user-tuned parameter that balances the size of margin and the training error. Note that we can use the same optimization technique (i.e., convex quadratic programming) to solve Eq. (7.16) as for hard-margin linear SVM. Likewise, the resulting soft-margin linear classifier uses the same equation (Eq. (7.15)) to classify a test tuple.

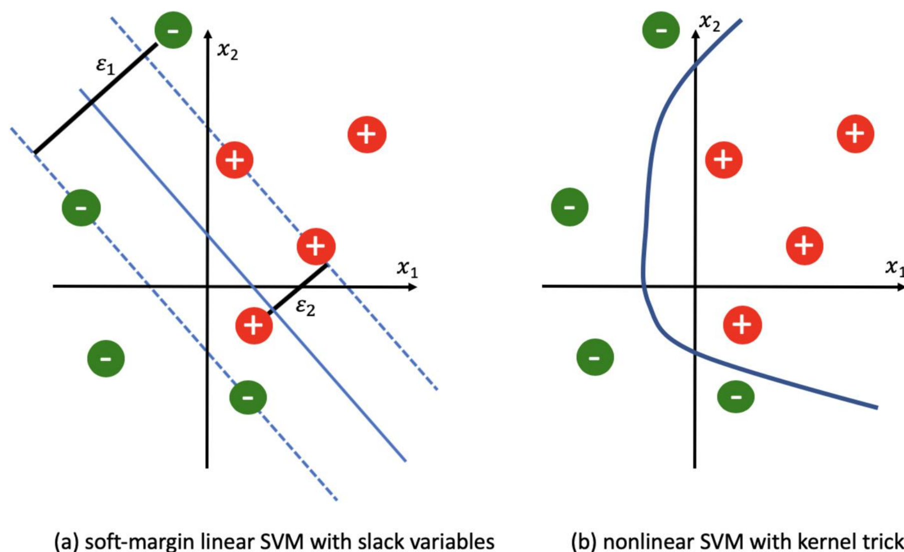


FIGURE 7.8

A simple 2-D case showing linearly inseparable data, where each tuple is represented by two attributes (x_1 and x_2). Unlike the linearly separable data of Fig. 7.5, here it is not possible to draw a straight line to perfectly separate the two classes. We could use a soft-margin linear SVM, with the help of slack variables (ϵ_1 and ϵ_2), to produce a linear decision boundary at the expense of two training tuples being mis-classified (a). Alternatively, we could seek for a nonlinear decision boundary (b).

$$\min \|\mathbf{W}\|^2 + C\sum_{i=1}^{N}\xi_i,$$
$$\text{s.t.} \quad y_i(\mathbf{W}'\mathbf{X}_i + b) \geq 1 - \xi_i, \quad \forall i. \tag{7.16}$$

We end this section with two important things to note. The complexity of the learned classifier is characterized by the number of support vectors rather than the dimensionality of the data. Hence SVMs tend to be less prone to overfitting than some other methods. The support vectors are the essential or critical training tuples—they lie closest to the decision boundary (MMH). If all other training tuples were removed and training were repeated, the same separating hyperplane would be found. Furthermore, the number of support vectors found can be used to compute an (upper) bound on the expected error rate of the SVM classifier, which is independent of the data dimensionality. An SVM with a small number of support vectors can have good generalization, even when the dimensionality of the data is high.

7.3.2 Nonlinear support vector machines

In Section 7.3.1, we learned about hard-margin linear SVMs for classifying linearly separable data. We also learned about soft-margin linear SVMs when the training data are linearly inseparable, by allowing

a small fraction of training tuples to be mis-classified. However, what if we want a "better" classifier to avoid such mis-classifications? For linearly inseparable cases (e.g., Fig. 7.8), no straight line can be found that would perfectly separate the classes.

The good news is that the approaches described for linear SVMs with both hard-margin and soft margin can be extended to create *nonlinear SVMs* for the classification of *linearly inseparable data* (also called *nonlinearly separable data*, or *nonlinear data* for short). Such SVMs are capable of finding nonlinear decision boundaries (i.e., nonlinear hypersurfaces) in input space.

"*So*," you may ask, "*how can we extend the linear approach?*" We obtain a nonlinear SVM by extending the approach for linear SVMs as follows. There are two main steps. In the first step, we transform the original input data into a higher dimensional space using a nonlinear mapping. Several common nonlinear mappings can be used in this step, as we will further describe next. Once the data have been transformed into the new higher dimensional space, the second step searches for a linear separating hyperplane in the new space. We again end up with an optimization problem that can be solved using the linear SVM formulation (i.e., convex quadratic programming). The maximal margin hyperplane found in the new space corresponds to a nonlinear separating hypersurface in the original space.

Example 7.2. Nonlinear transformation of original input data into a higher dimensional space. Consider the following example. A 3-D input vector $\boldsymbol{X} = (x_1, x_2, x_3)$ is mapped into a 6-D space, $\boldsymbol{Z}$, using the mappings $\phi_1(\boldsymbol{X}) = x_1$, $\phi_2(\boldsymbol{X}) = x_2$, $\phi_3(\boldsymbol{X}) = x_3$, $\phi_4(\boldsymbol{X}) = (x_1)^2$, $\phi_5(\boldsymbol{X}) = x_1x_2$, and $\phi_6(\boldsymbol{X}) = x_1x_3$. A decision hyperplane in the new space is $d(\boldsymbol{Z}) = \boldsymbol{W}'\boldsymbol{Z} + b$, where $\boldsymbol{W}$ and $\boldsymbol{Z}$ are vectors. This is linear with respect to the new features $\boldsymbol{Z}$. We solve for $\boldsymbol{W}$ and b and then substitute back so that the linear decision hyperplane in the new ($\boldsymbol{Z}$) space corresponds to a nonlinear second-order polynomial in the original 3-D input space:

$$\begin{aligned} d(\boldsymbol{Z}) &= w_1x_1 + w_2x_2 + w_3x_3 + w_4(x_1)^2 + w_5x_1x_2 + w_6x_1x_3 + b \\ &= w_1z_1 + w_2z_2 + w_3z_3 + w_4z_4 + w_5z_5 + w_6z_6 + b. \end{aligned}$$

□

However, there are some problems. First, how do we choose the nonlinear mapping to a higher dimensional space? Second, the computation involved will be costly. Refer to Eq. (7.15) for the classification of a test tuple, $\boldsymbol{X}$. Given the test tuple, we have to compute its dot product with every one of the support vectors.[4] In training, we have to compute a similar dot product for each pair of training tuples in order to find the MMH. This is especially expensive. Hence, the dot product computation required is very heavy and costly. We need another trick!

Luckily, we can use another math trick. It so happens that in solving the quadratic optimization problem of the linear SVM (i.e., when searching for a linear SVM in the new higher dimensional space), the training tuples appear only in the form of dot products, $\phi(\boldsymbol{X}_i) \cdot \phi(\boldsymbol{X}_j)$, where $\phi(\boldsymbol{X})$ is simply the nonlinear mapping function applied to transform the training tuples. Instead of computing the dot product on the transformed data tuples, it turns out that it is mathematically equivalent to instead

[4] The dot product of two vectors, $\boldsymbol{X} = (x_1, x_2, \ldots, x_n)$ and $\boldsymbol{X}_i = (x_{i1}, x_{i2}, \ldots, x_{in})$ is $x_1x_{i1} + x_2x_{i2} + \cdots + x_nx_{in}$. Note that this involves one multiplication and one addition for each of the n dimensions.

applying a *kernel function*, $K(\boldsymbol{X}_i, \boldsymbol{X}_j)$, to the original input data. That is,

$$K(\boldsymbol{X}_i, \boldsymbol{X}_j) = \phi(\boldsymbol{X}_i) \cdot \phi(\boldsymbol{X}_j). \tag{7.17}$$

In other words, everywhere that $\phi(\boldsymbol{X}_i) \cdot \phi(\boldsymbol{X}_j)$ appears in the training algorithm, we can replace it with $K(\boldsymbol{X}_i, \boldsymbol{X}_j)$. In this way, all calculations are made in the original input space, which is of potentially much lower dimensionality! We can safely avoid the mapping—it turns out that we don't even have to know what the mapping is! We will talk more later about what kinds of functions can be used as kernel functions for this problem. After applying this trick, we can then proceed to find a maximal margin separating hyperplane. The procedure is similar to that described in Section 7.3.1.

Example 7.3. Fig. 7.9(a) shows a training set with four positive tuples and four negative tuples. In the original feature space, each tuple is represented by two features (x_1 and x_2), where the training set is linearly inseparable (Fig. 7.9(b)). If we transform the original feature space into a 3-D space: $\Phi_1 = x_1^2$, $\Phi_2 = x_2^2$ and $\Phi_3 = \sqrt{2}x_1x_2$. In the transformed feature space (Fig. 7.9(c)), the positive tuples are linearly separable from the negative tuples. In other words, we can use a hyperplane $\Phi_1 + \Phi_2 = 2.5$ to perfectly separate all positive tuples from all negative tuples. The hyperplane in the transformed feature space is equivalent to a nonlinear decision boundary in the original 2-D space $x_1^2 + x_2^2 = 2.5$. Note that the dot product of two tuples (X_i and X_j) in the transformed feature space can be computed directly from the original feature space: $\Phi(X_i) \cdot \Phi(X_i) = (X_i \cdot X_j)^2$. □

"*What are some of the kernel functions that could be used?*" Properties of the kinds of kernel functions that could be used to replace the dot product have been studied. Three admissible kernel functions are

Polynomial kernel of degree *h*: $K(\boldsymbol{X}_i, \boldsymbol{X}_j) = (\boldsymbol{X}_i \cdot \boldsymbol{X}_j + 1)^h$,

Gaussian radial basis function kernel: $K(\boldsymbol{X}_i, \boldsymbol{X}_j) = e^{-\|\boldsymbol{X}_i - \boldsymbol{X}_j\|^2/2\sigma^2}$,

Sigmoid kernel: $K(\boldsymbol{X}_i, \boldsymbol{X}_j) = \tanh(\kappa \boldsymbol{X}_i \cdot \boldsymbol{X}_j - \delta)$.

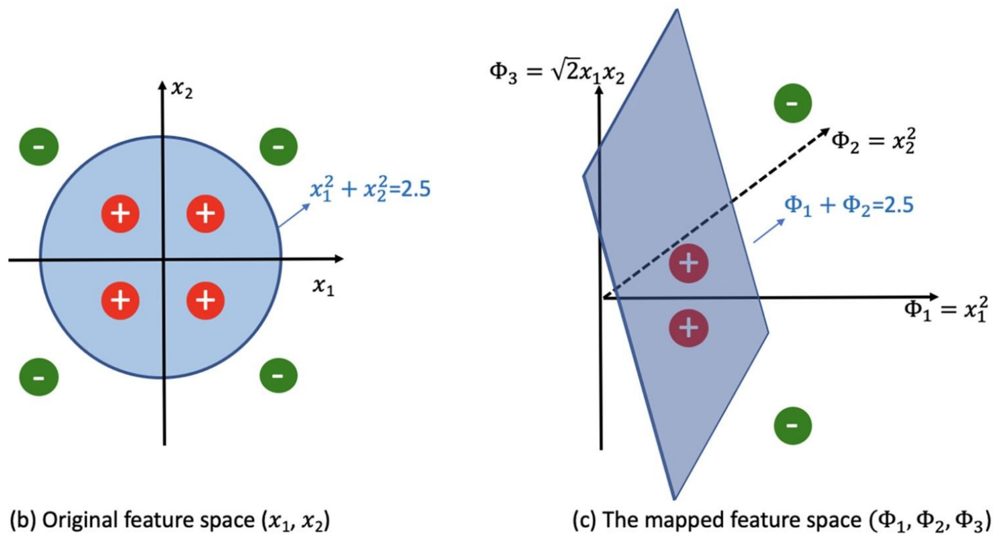
Each of these results in a different nonlinear classifier in (the original) input space. There are no golden rules for determining which admissible kernel will result in the most accurate SVM. In practice, the kernel chosen does not generally make a large difference in resulting accuracy.

So far, we have described linear and nonlinear SVMs for binary (i.e., two-class) classification. SVM classifiers can be combined for the multiclass case. See Section 7.7.1 for some strategies, such as training one classifier per class and the use of error-correcting codes.

A major research goal regarding SVMs is to improve the speed in training and testing so that SVMs may become a more feasible option for very large data sets (e.g., millions of support vectors). A very efficient strategy is to train SVMs in its prime form directly (e.g., Eqs. (7.14) and (7.16)) based on stochastic subgradient descent. Recall that in Chapter 7, we have used a similar technique called stochastic gradient descent to address the scalability issue of logistic regression classifier. Other issues include (1) determining the best kernel for a given data set and finding more efficient methods for the multiclass case, (2) making the SVMs more robust to the noise in the training data by using alternative norms of the weight vector **W** (e.g., l_1 norm SVM, $l_{2,1}$ norm SVM, capped l_p norm SVM). A key idea behind nonlinear SVM is the kernel trick, where we find a nonlinear classifier without explicitly constructing the nonlinear mapping. The kernel trick has been broadly applied to other data mining tasks, including regression, clustering, and so on.

	tuple index	1	2	3	4	5	6	7	8
Original features	x_1	1	1	-1	-1	2	-2	2	-2
	x_2	1	-1	1	-1	2	2	-2	-2
	Class label	+	+	+	+	-	-	-	-
Mapped features	x_1^2	1	1	1	1	4	4	4	4
	x_2^2	1	1	1	1	4	4	4	4
	$\sqrt{2}x_1x_2$	$\sqrt{2}$	$-\sqrt{2}$	$-\sqrt{2}$	$\sqrt{2}$	$4\sqrt{2}$	$-4\sqrt{2}$	$-4\sqrt{2}$	$4\sqrt{2}$

(a) Training tuples in the original feature space (x_1, x_2) and in the mapped feature space (shaded)



(b) Original feature space (x_1, x_2)

(c) The mapped feature space (Φ_1, Φ_2, Φ_3)

FIGURE 7.9

An example of kernel trick. (a) Training tuples in the original 2-D feature space and the transformed 3-D space (shaded). Training tuples in the original feature space are linearly inseparable (b), but become linearly separable in the transformed feature space (c). A linear decision boundary (i.e., a hyperplane) in the transformed feature space is equivalent to a nonlinear decision boundary in the original feature space. The dot product of two tuples in the transformed feature space can be computed directly from the original feature space.

7.4 Rule-based and pattern-based classification

In this section, we look at rule-based and pattern-based classifiers. For the former, the learned model is represented as a set of IF-THEN rules. We first examine how such rules are used for classification (Section 7.4.1). We then study ways in which they can be generated, either from a decision tree (Section 7.4.2) or directly from the training data using a *sequential covering algorithm* (Section 7.4.3). Based on that, we introduce pattern-based classifiers, where frequent patterns are used for classification. Section 7.4.4 explores **associative classification**, where association rules are generated from frequent

patterns and used for classification. The general idea is that we can search for strong associations between frequent patterns (conjunctions of attribute–value pairs) and class labels. Associative classification is a form of rule-based classifier, in that we often organize the mined association rule to form a rule-based classifier. Section 7.4.5 explores **discriminative frequent pattern–based classification**, where frequent patterns serve as combined features, which are considered in addition to single features when building a classification model. Because frequent patterns explore highly confident associations among multiple attributes, frequent pattern–based classification may overcome some constraints introduced by decision tree induction, which often only considers one attribute at a time. Studies have shown many frequent pattern–based classification methods to have greater accuracy and scalability than some traditional classification methods such as C4.5.

7.4.1 Using IF-THEN rules for classification

Rules are a good way of representing information or bits of knowledge. A **rule-based classifier** uses a set of IF-THEN rules for classification. An **IF-THEN** rule is an expression of the form

IF *condition* THEN *conclusion*.

An example is rule $R1$,

$$R1\text{: IF } age = youth \text{ AND } student = yes \text{ THEN } buys_computer = yes.$$

The “IF” part (or left side) of a rule is known as the **rule antecedent** or **precondition**. The “THEN” part (or right side) is the **rule consequent**. In the rule antecedent, the condition consists of one or more *attribute tests* (e.g., $age = youth$ and $student = yes$) that are logically ANDed. The rule’s consequent contains a class prediction (in this case, we are predicting whether a customer will buy a computer). $R1$ can also be written as

$$R1\text{: } (age = youth) \wedge (student = yes) \Rightarrow (buys_computer = yes).$$

If the condition (i.e., all the attribute tests) in a rule antecedent holds true for a given tuple, we say that the rule antecedent is **satisfied** (or simply, that the rule is satisfied) and that the rule **covers** the tuple.

A rule R can be assessed by its coverage and accuracy. Given a tuple, $\boldsymbol{X}$, from a class-labeled data set, D, let n_{covers} be the number of tuples covered by R; $n_{correct}$ be the number of tuples correctly classified by R; and $|D|$ be the number of tuples in D. We can define the **coverage** and **accuracy** of R as

$$coverage(R) = \frac{n_{covers}}{|D|} \tag{7.18}$$

$$accuracy(R) = \frac{n_{correct}}{n_{covers}}. \tag{7.19}$$

That is, a rule’s coverage is the percentage of tuples that are covered by the rule (i.e., their attribute values hold true for the rule’s antecedent). For a rule’s accuracy, we look at the tuples that it covers and see what percentage of them the rule can correctly classify.

Example 7.4. Rule accuracy and coverage. Let's go back to our data in Section 6.2, Table 6.1. These are class-labeled tuples from the *AllElectronics* customer database. Our task is to predict whether a customer will buy a computer. Consider rule $R1$, which covers 2 of the 14 tuples. It can correctly classify both tuples. Therefore $coverage(R1) = 2/14 = 14.28\%$ and $accuracy(R1) = 2/2 = 100\%$. □

Let's see how we can use rule-based classification to predict the class label of a given tuple, $\boldsymbol{X}$. If a rule is satisfied by $\boldsymbol{X}$, the rule is said to be **triggered**. For example, suppose we have

$$\boldsymbol{X} = (age = youth,\ income = medium,\ student = yes,\ credit_rating = fair).$$

We would like to classify $\boldsymbol{X}$ according to *buys_computer*. $\boldsymbol{X}$ satisfies $R1$, which triggers the rule.

If $R1$ is the only rule satisfied, then the rule **fires** by returning the class prediction for $\boldsymbol{X}$. Note that triggering does not always mean firing because there may be more than one rule that is satisfied! If more than one rule is triggered, we have a potential problem. What if each of them specifies a different class? Or what if no rule is satisfied by $\boldsymbol{X}$?

We tackle the first question. If more than one rule is triggered, we need a **conflict resolution strategy** to figure out which rule gets to fire and assign its class prediction to $\boldsymbol{X}$. There are many possible strategies. We look at two, namely *size ordering* and *rule ordering*.

The **size ordering** scheme assigns the highest priority to the triggering rule that has the "toughest" requirements, where toughness is measured by the rule antecedent *size*. That is, the triggering rule with the most attribute tests is fired.

The **rule ordering** scheme prioritizes the rules beforehand. The ordering may be *class-based* or *rule-based*. With **class-based ordering**, the classes are sorted in order of decreasing "importance" such as by decreasing *order of prevalence*. That is, all the rules for the most prevalent (or most frequent) class come first, the rules for the next prevalent class come next, and so on. Alternatively, they may be sorted based on the misclassification cost per class. Within each class, the rules are not ordered—they don't have to be because they all predict the same class (and so there can be no class conflict!).

With **rule-based ordering**, the rules are organized into one long priority list, according to some measure of rule quality, such as accuracy, coverage, size (number of attribute tests in the rule antecedent), or based on advice from domain experts. When rule ordering is used, the rule set is known as a **decision list**. With rule ordering, the triggering rule that appears earliest in the list has the highest priority, and so it gets to fire its class prediction. Any other rule that satisfies $\boldsymbol{X}$ is ignored. Most rule-based classification systems use a class-based rule-ordering strategy.

Note that in the first strategy, overall the rules are *unordered*. They can be applied in any order when classifying a tuple. That is, a disjunction (logical OR) is implied between different rules. Each rule represents a standalone nugget or piece of knowledge. This is in contrast to the rule ordering (decision list) scheme for which rules must be applied in the prescribed order so as to avoid conflicts. Each rule in a decision list implies the negation of the rules that come before it in the list. Hence rules in a decision list are more difficult to interpret.

Now that we have seen how we can handle conflicts, let's go back to the scenario where there is no rule satisfied by $\boldsymbol{X}$. How, then, can we determine the class label of $\boldsymbol{X}$? In this case, a fallback or **default rule** can be set up to specify a default class, based on a training set. This may be the class in majority or the majority class of the tuples that were not covered by any rule. The default rule is evaluated at the

end, if and only if no other rule covers X. The condition in the default rule is empty. In this way, the rule fires when no other rule is satisfied.

In the following sections, we examine how to build a rule-based classifier.

7.4.2 Rule extraction from a decision tree

In Section 6.2, we learned how to build a decision tree classifier from a set of training data. Decision tree classifiers are a popular method of classification—it is easy to understand how decision trees work and they are known for their accuracy. Decision trees can become large and difficult to interpret. In this subsection, we look at how to build a rule-based classifier by extracting IF-THEN rules from a decision tree. In comparison with a decision tree, the IF-THEN rules may be easier for humans to understand, particularly if the decision tree is very large.

To extract rules from a decision tree, one rule is created for each path from the root to a leaf node. Each splitting criterion along a given path is logically ANDed to form the rule antecedent ("IF" part). The leaf node holds the class prediction, forming the rule consequent ("THEN" part).

Example 7.5. Extracting classification rules from a decision tree. The decision tree of Fig. 6.2 can be converted to classification IF-THEN rules by tracing the path from the root node to each leaf node in the tree. The rules extracted from Fig. 6.2 are as follows:

$$
\begin{array}{llll}
R1: & \text{IF } age = youth & \text{AND } student = no & \text{THEN } buys_computer = no \\
R2: & \text{IF } age = youth & \text{AND } student = yes & \text{THEN } buys_computer = yes \\
R3: & \text{IF } age = middle_aged & & \text{THEN } buys_computer = yes \\
R4: & \text{IF } age = senior & \text{AND } credit_rating = excellent & \text{THEN } buys_computer = yes \\
R5: & \text{IF } age = senior & \text{AND } credit_rating = fair & \text{THEN } buys_computer = no.
\end{array}
$$

□

A disjunction (logical OR) is implied between each of the extracted rules. Because the rules are extracted directly from the tree, they are **mutually exclusive** and **exhaustive**. *Mutually exclusive* means that we cannot have rule conflicts here because no two rules will be triggered for the same tuple. (We have one rule per leaf, and any tuple can map to only one leaf.) *Exhaustive* means there is one rule for each possible attribute–value combination, so that this set of rules does not require a default rule. Therefore the order of the rules does not matter—they are *unordered*.

Since we end up with one rule per leaf, the set of extracted rules is not much simpler than the corresponding decision tree! The extracted rules may be even more difficult to interpret than the original trees in some cases. As an example, Fig. 6.7 shows decision trees that suffer from subtree repetition and replication. The resulting set of rules extracted can be large and difficult to follow, because some of the attribute tests may be irrelevant or redundant. So, the plot thickens. Although it is easy to extract rules from a decision tree, we may need to do some more work by pruning the resulting rule set.

"How can we prune the rule set?" For a given rule antecedent, any condition that does not improve the estimated accuracy of the rule can be pruned (i.e., removed), thereby generalizing the rule. C4.5 extracts rules from an unpruned tree, and then prunes the rules using a pessimistic approach similar to its tree pruning method. The training tuples and their associated class labels are used to estimate rule

accuracy. However, because this would result in an optimistic estimate, alternatively, the estimate is adjusted to compensate for the bias, resulting in a pessimistic estimate. In addition, any rule that does not contribute to the overall accuracy of the entire rule set can also be pruned.

Other problems arise during rule pruning, however, as the rules *will no longer be* mutually exclusive and exhaustive. For conflict resolution, C4.5 adopts a **class-based ordering scheme**. It groups together all rules for a single class, and then determines a ranking of these class rule sets. Within a rule set, the rules are not ordered. C4.5 orders the class rule sets to minimize the number of *false-positive errors* (i.e., where a rule predicts a class, C, but the actual class is not C). The class rule set with the least number of false positives is examined first. Once pruning is complete, a final check is done to remove any duplicates. When choosing a default class, C4.5 does not choose the majority class, because this class will likely have many rules for its tuples. Instead, it selects the class that contains the most training tuples that were not covered by any rule.

7.4.3 Rule induction using a sequential covering algorithm

IF-THEN rules can be extracted directly from the training data (i.e., without having to generate a decision tree first) using a **sequential covering algorithm**. The name comes from the notion that the rules are learned *sequentially* (one at a time), where each rule for a given class will ideally *cover* many of the class's tuples (and hopefully none of the tuples of other classes). Sequential covering algorithms are the most widely used approach to mining disjunctive sets of classification rules and form the topic of this subsection.

There are many sequential covering algorithms. Popular variations include AQ, CN2, and the more recent RIPPER. The general strategy is as follows. Rules are learned one at a time. Each time a rule is learned, the tuples covered by the rule are removed, and the process repeats on the remaining tuples. This sequential learning of rules is in contrast to decision tree induction. Because the path to each leaf in a decision tree corresponds to a rule, we can consider decision tree induction as learning a set of rules *simultaneously*.

A basic sequential covering algorithm is shown in Fig. 7.10. Here, rules are learned for one class at a time. Ideally, when learning a rule for a class, C, we would like the rule to cover all (or as many as possible) of the training tuples of class C and none (or as few as possible) of the tuples from other classes. In this way, the rules learned should be of high accuracy. The rules need not necessarily be of high coverage. This is because we can have more than one rule for a class, so that different rules may cover different tuples within the same class. The process continues until the terminating condition is met, such as when there are no more training tuples or the quality of a rule returned is below a user-specified threshold. The *Learn_One_Rule* procedure finds the "best" rule for the current class, given the current set of training tuples.

"How are rules learned?" Typically, rules are grown in a *general-to-specific* manner (Fig. 7.11). We can think of this as a beam search, where we start off with an empty rule and then gradually keep appending attribute tests to it. We append by adding the attribute test as a logical conjunction to the existing condition of the rule antecedent. Suppose our training set, D, consists of loan application data. Attributes regarding each applicant include their age, income, education level, residence, credit rating, and the term of the loan. The classifying attribute is *loan_decision*, which indicates whether a loan is accepted (considered safe) or rejected (considered risky). To learn a rule for the class "accept," we start

Algorithm: Sequential covering. Learn a set of IF-THEN rules for classification.

Input:

- D, a data set of class-labeled tuples;
- Att_vals, the set of all attributes and their possible values.

Output: A set of IF-THEN rules.
Method:

(1) $Rule_set = \{\}$; // initial set of rules learned is empty
(2) **for each** class c **do**
(3) **repeat**
(4) Rule = **Learn_One_Rule**(D, Att_vals, c);
(5) remove tuples covered by $Rule$ from D;
(6) $Rule_set = Rule_set + Rule$; // add new rule to rule set
(7) **until** terminating condition;
(8) **endfor**
(9) return $Rule_Set$;

FIGURE 7.10

Basic sequential covering algorithm.

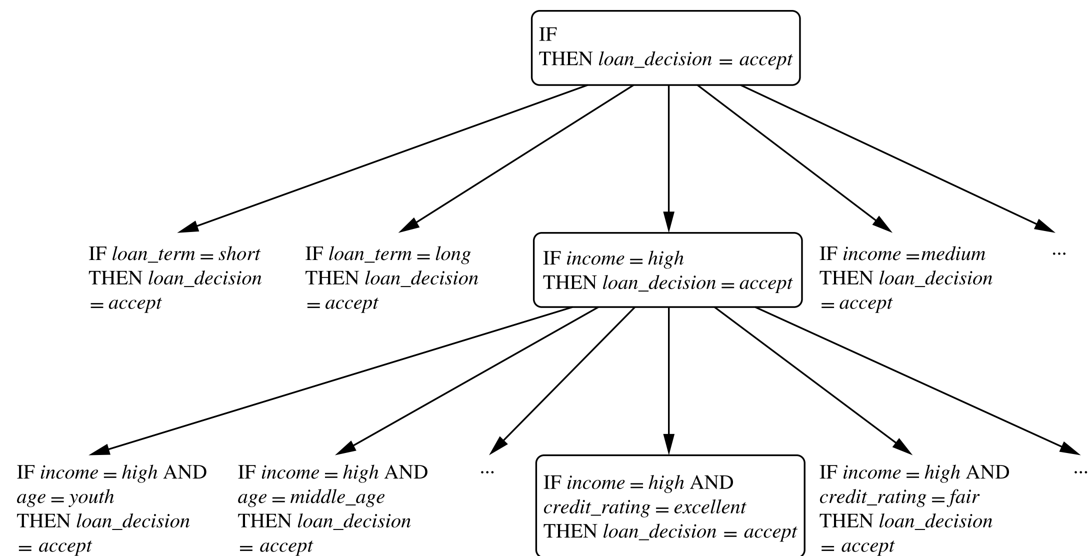


FIGURE 7.11

A general-to-specific search through rule space.

off with the most general rule possible, that is, the condition of the rule antecedent is empty. The rule is

$$\text{IF THEN } loan_decision = accept.$$

We then consider each possible attribute test that may be added to the rule. These can be derived from the parameter *Att_vals*, which contains a list of attributes with their associated values. For example, for an attribute–value pair (att, val), we can consider attribute tests such as $att = val$, $att \leq val$, $att > val$, and so on. Typically, the training data will contain many attributes, each of which may have several possible values. Finding an optimal rule set becomes computationally explosive. Instead, *Learn_One_Rule* adopts a greedy depth-first strategy. Each time it is faced with adding a new attribute test (conjunction) to the current rule, it picks the one that improves the rule quality most, based on the training samples. We will say more about rule quality measures in a minute. For now, let's say we use rule accuracy as our quality measure. Getting back to our example with Fig. 7.11, suppose *Learn_One_Rule* finds that the attribute test *income* = *high* best improves the accuracy of our current (empty) rule. We append it to the condition, so that the current rule becomes

IF *income* = *high* THEN *loan_decision* = *accept*.

Each time we add an attribute test to a rule, the resulting rule should cover relatively more of the "accept" tuples. During the next iteration, we again consider the possible attribute tests and end up selecting *credit_rating* = *excellent*. Our current rule grows to become

IF *income* = *high* AND *credit_rating* = *excellent* THEN *loan_decision* = *accept*.

The process repeats, where at each step we continue to greedily grow rules until the resulting rule meets an acceptable quality level.

Greedy search does not allow for backtracking. At each step, we *heuristically* add what appears to be the best choice at the moment. What if we unknowingly made a poor choice along the way? To lessen the chance of this happening, instead of selecting the best attribute test to append to the current rule, we can select the best k attribute tests. In this way, we perform a beam search of width k, wherein we maintain the k best candidates overall at each step, rather than a single best candidate.

Rule quality measures

Learn_One_Rule needs a measure of rule quality. Every time it considers an attribute test, it must check to see if appending such a test to the current rule's condition will result in an improved rule. Accuracy may seem like an obvious choice at first, but consider Example 7.6.

Example 7.6. Choosing between two rules based on accuracy. Consider the two rules as illustrated in Fig. 7.12. Both are for the class *loan_decision* = *accept*. We use "*a*" to represent the tuples of class "*accept*" and "*r*" for the tuples of class "*reject*." Rule $R1$ correctly classifies 38 of the 40 tuples it covers. Rule $R2$ covers only two tuples, which it correctly classifies. Their respective accuracies are 95% and 100%. Thus $R2$ has greater accuracy than $R1$, but it is not the better rule because of its small coverage. □

From this example, we see that accuracy on its own is not a reliable estimate of rule quality. Coverage on its own is not useful either—for a given class we could have a rule that covers many tuples, most of which belong to other classes! Thus we seek other measures for evaluating rule quality, which may integrate aspects of accuracy and coverage. Here we will look at some, namely *entropy*, another based on *information gain*, and a *statistical test* that considers coverage. For our discussion, suppose we are learning rules for the class c. Our current rule is R: IF $condition$ THEN $class = c$. We want to see

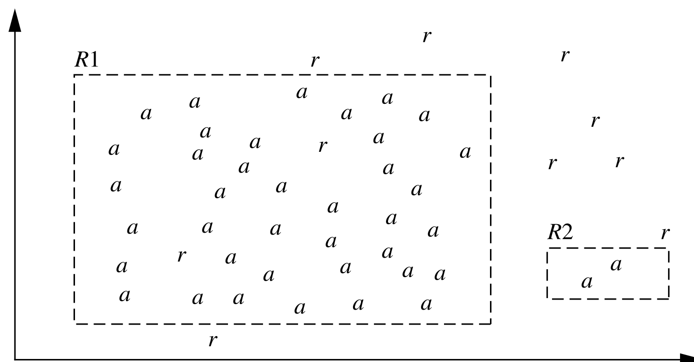


FIGURE 7.12

Rules for the class *loan_decision = accept*, showing *accept* (a) and *reject* (r) tuples.

if logically ANDing a given attribute test to $condition$ would result in a better rule. We call the new condition, $condition'$, where R': IF $condition'$ THEN $class = c$ is our potential new rule. In other words, we want to see if R' is any better than R.

We have already seen entropy in our discussion of the information gain measure used for attribute selection in decision tree induction. It is also known as the *expected information* needed to classify a tuple in data set, D. Here, D is the set of tuples covered by $condition'$ and p_i is the probability of class C_i in D. The lower the entropy, the better $condition'$ is. Entropy prefers conditions that cover a large number of tuples of a single class and few tuples of other classes.

Another measure is based on information gain and was proposed in **FOIL** (First-Order Inductive Learner), a sequential covering algorithm that learns first-order logic rules. Learning first-order rules is more complex because such rules contain variables, whereas the rules we are concerned with in this section are propositional (i.e., variable-free).[5] In machine learning, the tuples of the class for which we are learning rules are called *positive* tuples, whereas the remaining tuples are *negative*. Let *pos* and *neg* be the number of positive and negative tuples covered by R, respectively. Let *pos'* and *neg'* be the number of positive (negative) tuples covered by R', respectively. FOIL assesses the information gained by extending $condition'$ as

$$FOIL_Gain = pos' \times \left(\log_2 \frac{pos'}{pos' + neg'} - \log_2 \frac{pos}{pos + neg}\right). \tag{7.20}$$

It favors rules that have high accuracy and cover many positive tuples.

We can also use a statistical test of significance to determine if the apparent effect of a rule is not attributed to chance but instead indicates a genuine correlation between attribute values and classes. The test compares the observed distribution among classes of tuples covered by a rule with the expected distribution that would result if the rule made predictions at random. We want to assess whether any

5 Incidentally, FOIL was also proposed by Quinlan, the father of ID3.

observed differences between these two distributions may be attributed to chance. We can use the **likelihood ratio statistic**,

$$Likelihood_Ratio = 2\sum_{i=1}^{m} f_i \log\left(\frac{f_i}{e_i}\right), \tag{7.21}$$

where m is the number of classes.

For tuples satisfying the rule, f_i is the observed frequency of each class i among the tuples. e_i is what we would expect the frequency of each class i to be if the rule made random predictions. The statistic has a χ^2 distribution with $m-1$ degrees of freedom. The higher the likelihood ratio, the more likely that there is a *significant* difference in the number of correct predictions made by our rule in comparison with a "random guesser." That is, the performance of our rule is not due to chance. The ratio helps identify rules with insignificant coverage.

CN2 uses entropy together with the likelihood ratio test, while FOIL's information gain is used by RIPPER.

Rule pruning

Learn_One_Rule does not employ a test set when evaluating rules. Assessments of rule quality as described previously are made with tuples from the original training data. These assessments are optimistic because the rules will likely overfit the data. That is, the rules may perform well on the training data but less well on subsequent unseen data (i.e., test data). To compensate for this, we can prune the rules. A rule is pruned by removing a conjunction (attribute test). We choose to prune a rule, R, if the pruned version of R has greater quality, as assessed on an independent set of tuples. As in decision tree pruning, we refer to this set as a *pruning set*.

FOIL uses a simple yet effective method. Given a rule, R,

$$FOIL_Prune(R) = \frac{pos - neg}{pos + neg}, \tag{7.22}$$

where pos and neg are the number of positive and negative tuples covered by R, respectively. This value will increase with the accuracy of R on a pruning set. Therefore if the $FOIL_Prune$ value is higher for the pruned version of R, then we prune R.

By convention, RIPPER starts with the most recently added conjunction when considering pruning. Conjunctions are pruned one at a time as long as this results in an improvement.

7.4.4 Associative classification

In this section, you will learn about associative classification. The methods discussed are CBA, CMAR, and CPAR.

Before we begin, however, let's look at association rule mining in general. Association rules are mined in a two-step process consisting of *frequent itemset mining* followed by *rule generation*. The first step searches for patterns of attribute–value pairs that occur repeatedly in a data set, where each attribute–value pair is considered an *item*. The resulting attribute–value pairs form *frequent itemsets* (also referred to as *frequent patterns*). The second step analyzes the frequent itemsets to generate association rules. All association rules must satisfy certain criteria regarding their "accuracy" (or *confidence*)

and the proportion of the data set that they actually represent (referred to as *support*). For example, the following is an association rule mined from a data set, D, shown with its confidence and support:

$$age = youth \wedge credit = OK \Rightarrow buys_computer = yes$$
$$[support = 20\%, confidence = 93\%], \tag{7.23}$$

where $\wedge$ represents a logical "AND." We will say more about confidence and support later.

More formally, let D be a data set of tuples. Each tuple in D is described by n attributes, $A_1, A_2, \ldots, A_n$, and a class label attribute, A_{class}. All continuous attributes are discretized and treated as categorical (or nominal) attributes. An **item**, p, is an attribute–value pair of the form (A_i, v), where A_i is an attribute taking a value, v. A data tuple $\boldsymbol{X} = (x_1, x_2, \ldots, x_n)$ satisfies an item, $p = (A_i, v)$, if and only if $x_i = v$, where x_i is the value of the ith attribute of $\boldsymbol{X}$. Association rules can have any number of items in the rule antecedent (left side) and any number of items in the rule consequent (right side). However, when mining association rules for use in classification, we are only interested in association rules of the form $p_1 \wedge p_2 \wedge \ldots p_l \Rightarrow A_{class} = C$, where the rule antecedent is a conjunction of items, $p_1, p_2, \ldots, p_l$ $(l \leq n)$, associated with a class label, C. For a given rule, R, the percentage of tuples in D satisfying the rule antecedent that also has the class label C is called the **confidence** of R.

From a classification point of view, this is akin to rule accuracy. For example, a confidence of 93% for Rule in Eq. (7.23) means that 93% of the customers in D who are young and have an OK credit rating belong to the class *buys_computer* = *yes*. The percentage of tuples in D satisfying the rule antecedent and having class label C is called the **support** of R. A support of 20% for Rule in Eq. (7.23) means that 20% of the customers in D are young, have an OK credit rating, and belong to the class *buys_computer* = *yes*.

In general, associative classification consists of the following steps:

1. Mine the data for frequent itemsets, that is, find commonly occurring attribute–value pairs in the data.
2. Analyze the frequent itemsets to generate association rules per class, which satisfy confidence and support criteria.
3. Organize the rules to form a rule-based classifier.

Methods of associative classification differ primarily in the approach used for frequent itemset mining and in how the derived rules are analyzed and used for classification. We now look at some of the various methods for associative classification.

One of the earliest and simplest algorithms for associative classification is **CBA** (Classification Based on Associations). CBA uses an iterative approach to frequent itemset mining, similar to that described for Apriori in Section 4.2.1, where multiple passes are made over the data and the derived frequent itemsets are used to generate and test longer itemsets. In general, the number of passes made is equal to the length of the longest rule found. The complete set of rules satisfying minimum confidence and minimum support thresholds are found and then analyzed for inclusion in the classifier. CBA uses a heuristic method to construct the classifier, where the rules are ordered according to decreasing precedence based on their confidence and support. If a set of rules has the same antecedent, then the rule with the highest confidence is selected to represent the set. When classifying a new tuple, the first rule satisfying the tuple is used to classify it. The classifier also contains a default rule, having the lowest precedence, which specifies a default class for any new tuple that is not satisfied by any other rule in

the classifier. In this way, the set of rules making up the classifier form a *decision list*. In general, CBA was empirically found to be more accurate than C4.5 on a good number of data sets.

CMAR (Classification based on Multiple Association Rules) differs from CBA in its strategy for frequent itemset mining and its construction of the classifier. It also employs several rule pruning strategies with the help of a tree structure for efficient storage and retrieval of rules. CMAR adopts a variant of the *FP-growth* algorithm to find the complete set of rules satisfying the minimum confidence and minimum support thresholds. FP-growth was described in Section 4.2.4. FP-growth uses a tree structure, called an *FP-tree*, to register all the frequent itemset information contained in the given data set, D. This requires only two scans of D. The frequent itemsets are then mined from the FP-tree. CMAR uses an enhanced FP-tree that maintains the distribution of class labels among tuples satisfying each frequent itemset. In this way, it is able to combine rule generation together with frequent itemset mining in a single step.

CMAR employs another tree structure to store and retrieve rules efficiently and to prune rules based on confidence, correlation, and database coverage. Rule pruning strategies are triggered whenever a rule is inserted into the tree. For example, given two rules, $R1$ and $R2$, if the antecedent of $R1$ is more general than that of $R2$ and $conf(R1) \geq conf(R2)$, then $R2$ is pruned. The rationale is that highly specialized rules with low confidence can be pruned if a more generalized version with higher confidence exists. CMAR also prunes rules for which the rule antecedent and class are not positively correlated, based on an χ^2 test of statistical significance.

"*If more than one rule applies, which one do we use?*" As a classifier, CMAR operates differently than CBA. Suppose that we are given a tuple $\boldsymbol{X}$ to classify and that only one rule satisfies or matches $\boldsymbol{X}$.[6] This case is trivial—we simply assign the rule's class label. Suppose, instead, that more than one rule satisfies $\boldsymbol{X}$. These rules form a set, S. Which rule would we use to determine the class label of $\boldsymbol{X}$? CBA would assign the class label of the most confident rule among the rule set, S. CMAR instead considers multiple rules when making its class prediction. It divides the rules into groups according to class labels. All rules within a group share the same class label and each group has a distinct class label.

CMAR uses a weighted χ^2 measure to find the "strongest" group of rules, based on the statistical correlation of rules within a group. It then assigns $\boldsymbol{X}$ the class label of the strongest group. In this way it considers multiple rules, rather than a single rule with highest confidence, when predicting the class label of a new tuple. In experiments, CMAR had slightly higher average accuracy in comparison with CBA. Its runtime, scalability, and use of memory were found to be more efficient.

"*Is there a way to cut down on the number of rules generated?*" CBA and CMAR adopt methods of frequent itemset mining to generate *candidate* association rules, which include all conjunctions of attribute–value pairs (items) satisfying minimum support. These rules are then examined, and a subset is chosen to represent the classifier. However, such methods generate quite a large number of rules. **CPAR** (Classification based on Predictive Association Rules) takes a different approach to rule generation, based on FOIL (a rule generation algorithm for classification). FOIL builds rules to distinguish positive tuples (e.g., $buys_computer = yes$) from negative tuples (e.g., $buys_computer = no$). For multiclass problems, FOIL is applied to each class. That is, for a class, C, all tuples of class C are considered positive tuples, while the rest are considered negative tuples. Rules are generated to distinguish C tuples from all others. Each time a rule is generated, the positive samples it satisfies (or *covers*) are

[6] If a rule's antecedent satisfies or matches $\boldsymbol{X}$, then we say that the rule satisfies $\boldsymbol{X}$.

removed until all the positive tuples in the data set are covered. In this way, fewer rules are generated. CPAR relaxes this step by allowing the covered tuples to remain under consideration, but reducing their weight. The process is repeated for each class. The resulting rules are merged to form the classifier rule set.

During classification, CPAR employs a somewhat different multirule strategy than CMAR. If more than one rule satisfies a new tuple, $\boldsymbol{X}$, the rules are divided into groups according to class, similar to CMAR. However, CPAR uses the best k rules of each group to predict the class label of $\boldsymbol{X}$, based on expected accuracy. By considering the best k rules rather than all of a group's rules, it avoids the influence of lower-ranked rules. CPAR's accuracy on numerous data sets was shown to be close to that of CMAR. However, since CPAR generates far fewer rules than CMAR, it shows much better efficiency with large sets of training data.

In summary, associative classification offers an alternative classification scheme by building rules based on conjunctions of attribute–value pairs that occur frequently in data.

7.4.5 Discriminative frequent pattern–based classification

From work on associative classification, we see that frequent patterns reflect strong associations between attribute–value pairs (or items) in data and are useful for classification.

"*But just how discriminative are frequent patterns for classification?*" Frequent patterns represent feature combinations. Let's compare the discriminative power of frequent patterns and single features. Fig. 7.13 plots the information gain of frequent patterns and single features (i.e., of pattern length 1) for three UCI data sets.[7] The discrimination power of some frequent patterns is higher than that of single features. Frequent patterns map data to a higher-dimensional space. They capture more underlying semantics of the data and thus can hold greater expressive power than single features.

"*Why not consider frequent patterns as combined features, in addition to single features when building a classification model?*" This notion is the basis of **frequent pattern–based classification**—the learning of a classification model in the feature space of single attributes *as well as* frequent patterns. In this way, we transfer the original feature space to a larger space. This will likely increase the chance of including important features.

Let's get back to our earlier question: How discriminative are frequent patterns? Many of the frequent patterns generated in frequent itemset mining are indiscriminative because they are solely based on support, without considering predictive power. That is, by definition, a pattern must satisfy a user-specified minimum support threshold, min_sup, to be considered frequent. For example, if min_sup is 5%, a pattern is frequent if it occurs in 5% of the data tuples. Consider Fig. 7.14, which plots information gain vs. pattern frequency (support) for three UCI data sets. A theoretical upper bound on information gain, which was derived analytically, is also plotted. The figure shows that the discriminative power (assessed here as information gain) of low-frequency patterns is bounded by a small value. This is due to the patterns' limited coverage of the data set. Similarly, the discriminative power of very high-frequency patterns is also bounded by a small value, which is due to their commonness in the data. The upper bound of information gain is a function of pattern frequency. These observations can

[7] The University of California at Irvine (UCI) archives several large data sets at http://kdd.ics.uci.edu/. These are commonly used by researchers for the testing and comparison of machine learning and data mining algorithms.

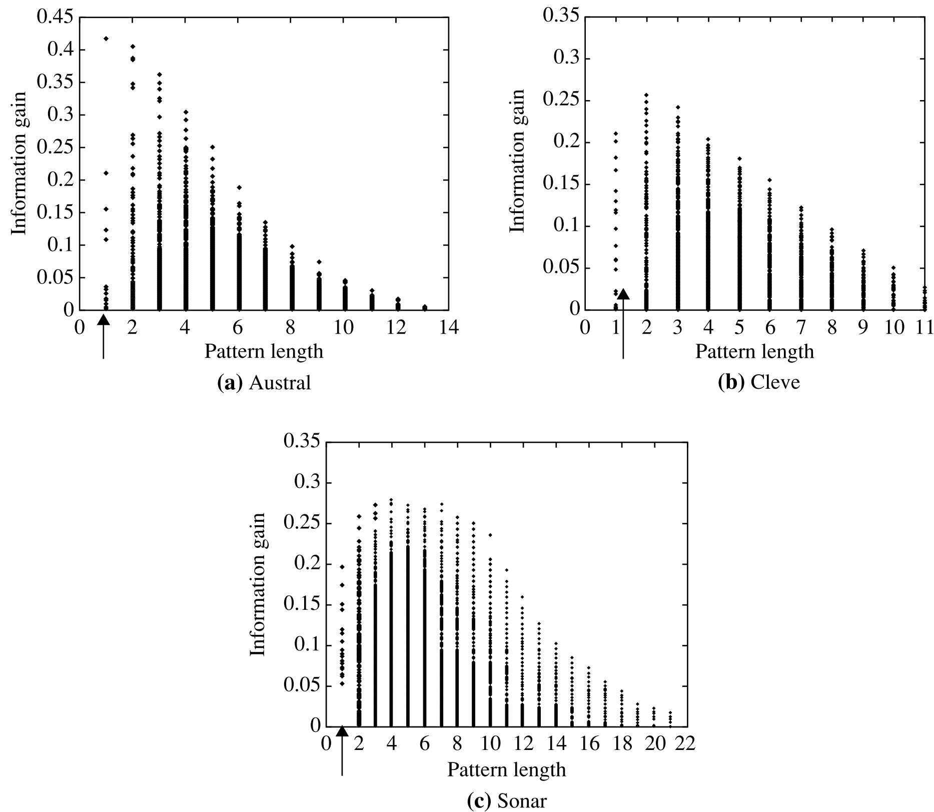


FIGURE 7.13

Single feature vs. frequent pattern: Information gain is plotted for single features (patterns of length 1, indicated by arrows) and frequent patterns (combined features) for three UCI data sets. *Source:* Adapted from Cheng, Yan, Han, and Hsu [CYHH07].

be confirmed analytically. Patterns with medium-large supports (e.g., $support = 300$ in Fig. 7.14(a)) may be discriminative or not. Thus not every frequent pattern is useful.

If we were to add all the frequent patterns to the feature space, the resulting feature space would be huge. This slows down the model learning process and may also lead to decreased accuracy due to a form of overfitting in which there are too many features. Many of the patterns may be also redundant. Therefore, it's a good idea to apply feature selection to eliminate the less discriminative and redundant frequent patterns as features. The *general framework for discriminative frequent pattern–based classification* is as follows.

1. **Feature generation:** The data, D, are partitioned according to class label. Use frequent itemset mining to discover frequent patterns in each partition, satisfying minimum support. The collection of frequent patterns, $\mathcal{F}$, makes up the feature candidates.

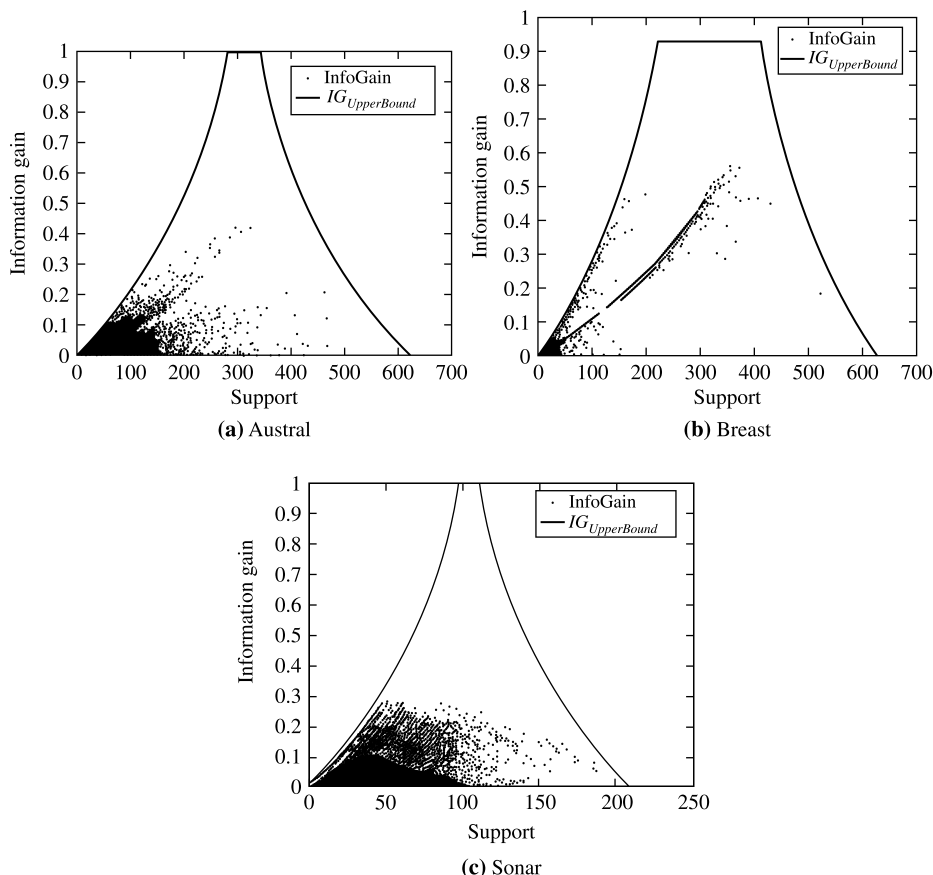


FIGURE 7.14

Information gain vs. pattern frequency (support) for three UCI data sets. A theoretical upper bound on information gain ($IG_{UpperBound}$) is also shown. *Source:* Adapted from Cheng, Yan, Han, and Hsu [CYHH07].

2. **Feature selection:** Apply feature selection to $\mathcal{F}$, resulting in $\mathcal{F}_\mathcal{S}$, the set of selected (more discriminating) frequent patterns. Information gain, Fisher score, or other evaluation measures can be used for this step. Relevancy checking can also be incorporated into this step to weed out redundant patterns. The data set D is transformed to D', where the feature space now includes the single features and the selected frequent patterns, $\mathcal{F}_\mathcal{S}$. Commonly used feature selection methods were introduced in Section 7.1.
3. **Learning of classification model:** A classifier is built on the data set D'. Any learning algorithm can be used as the classification model.

The general framework is summarized in Fig. 7.15(a), where the discriminative patterns are represented by dark circles. Although the approach is straightforward, we can encounter a computational

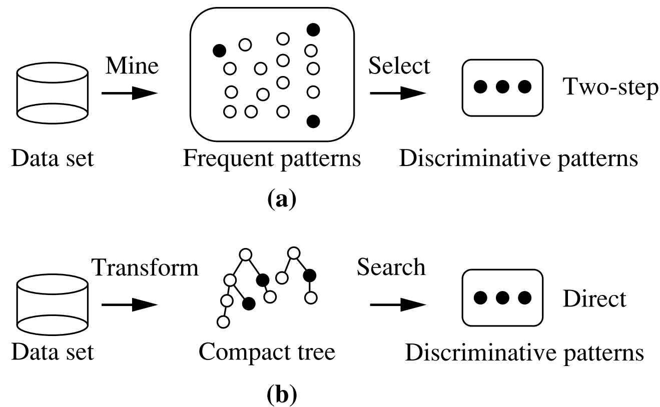


FIGURE 7.15

A framework for frequent pattern–based classification: (a) a two-step general approach vs. (b) the direct approach of DDPMine.

bottleneck by having to first find *all* the frequent patterns and then analyze *each one* for selection. The amount of frequent patterns found can be huge due to the explosive number of pattern combinations between items.

To improve the efficiency of the general framework, consider condensing steps 1 and 2 into just one step. That is, rather than generating the complete set of frequent patterns, it's possible to mine only the highly discriminative ones. This more direct approach is referred to as **DDPMine** (Direct Discriminative Pattern Mining). The DDPMine algorithm follows this approach, as illustrated in Fig. 7.15(b). It first transforms the training data into a compact tree structure known as a frequent pattern tree, or FP-tree (Chapter 4), which holds all of the attribute–value (itemset) association information. It then searches for discriminative patterns on the tree. The approach is direct in that it avoids generating a large number of indiscriminative patterns. It incrementally reduces the problem by eliminating training tuples, thereby progressively shrinking the FP-tree. This further speeds up the mining process.

By choosing to transform the original data to an FP-tree, DDPMine avoids generating redundant patterns because an FP-tree stores only the *closed* frequent patterns. By definition, any subpattern, β, of a closed pattern, α, is redundant with respect to α (Chapter 5). DDPMine directly mines the discriminative patterns and integrates feature selection into the mining framework. The theoretical upper bound on information gain is used to facilitate a branch-and-bound search, which prunes the search space significantly. Experimental results show that DDPMine achieves orders of magnitude speedup over the two-step approach without decline in classification accuracy. DDPMine also outperforms state-of-the-art associative classification methods in terms of both accuracy and efficiency.

Compared with associative classifiers, DDPMine is able to prune a huge number of nondiscriminative frequent patterns. However, DDPMine might still use hundreds or even thousands of frequent patterns in the classification model. *How can we further reduce the number of patterns to build a more compact classifier?* This will not only speed up the computation but also make the classifier more explainable to the end users. **DPClass** (Discriminative Pattern-based Classification) addresses this issue by combining the strength of two methods, including tree-based classifiers (e.g., decision tree clas-

sifier, random forest, etc., which were introduced in Section 6.2) and feature selection (e.g., forward selection, LASSO, etc., which were introduced in Section 7.1). DPClass works as follows. First, it uses random forest that contains multiple tree-based classifiers. Then, each prefix path from the root of a tree in random forest to its nonleaf node is treated as a discriminative pattern. Finally, it leverages feature selection, including forward feature selection and LASSO based method, to select a small subset of highly discriminative patterns to construct a linear classifier, such as logistic regression classifier or linear SVMs. The empirical evaluations on a good number of UCI data sets show that DPClass performs similarly as or better than DDPMine. On the other hand, DPClass uses a significantly less number of discriminative patterns than DDPMine. Therefore the classifier generated by DPClass is more compact, making itself faster in test and more explainable to end users.

7.5 Classification with weak supervision

The effectiveness of the classifiers we have introduced so far (e.g., SVMs, logistic regression, *k*-NN) largely depends on "strong supervision." It means that in order to train a highly accurate classifier, we typically need a large number of high-quality training tuples, and the true class label for each training tuple is accurately annotated, say by the domain experts. However, what if there is only a small number of labeled training tuples? Document classification, speech recognition, computer vision, and information extraction are just a few examples of applications in which unlabeled data are abundant. Consider *document classification*, for example. Suppose we want to build a model to automatically classify text documents like articles or web pages. In particular, we want the model to distinguish between hockey and football documents. We have a vast amount of documents available, yet the documents are not class-labeled. Recall that supervised learning requires a training set, that is, a set of class-labeled data. To have a human examine and assign a class label to individual documents (to form a training set) is time consuming and expensive. *Speech recognition* requires the accurate labeling of speech utterances by trained linguists. It was reported that 1 minute of speech takes 10 minutes to label, and annotating phonemes (basic units of sound) can take 400 times as long. *Information extraction systems* are trained using labeled documents with detailed annotations. These are obtained by having human experts highlight items or relations of interest in text such as the names of companies or individuals. High-level expertise may be required for certain knowledge domains such as gene and disease mentions in biomedical information extraction. Clearly, the manual assignment of class labels to prepare a training set can be extremely costly, time consuming, and tedious. In computer vision, a fundamental task is to build a highly accurate classifier to automatically recognize various objects (i.e., class labels). However, some objects (e.g., a new type of dog) might appear only *after* the classifier has been built. In other words, there are no training tuples at all for the newly appeared class label. How can the classifier still recognize the test image of such a new type of dog?

We study five approaches for classification that are suitable for situations where there is only a limited number or no labeled training tuples. Section 7.5.1 introduces *semisupervised classification*, which builds a classifier using both labeled and unlabeled data. Section 7.5.2 presents *active learning*, where the learning algorithm carefully selects a few of the unlabeled data tuples and asks a human to label only those tuples. Section 7.5.3 presents *transfer learning*, which aims to extract the knowledge from one or more source classification tasks (e.g., classifying camera reviews) and apply the knowledge to a target classification task (e.g., classifying TV reviews). Section 7.5.4 studies *distant supervision*

whose key idea is to automatically obtain a large number of inexpensive, but potentially noisy labeled training tuples. Finally, Section 7.5.5 introduces *zero-shot learning*, which deals with the case there are no training tuples for certain class labels at all. Each of these strategies can reduce the need to annotate large amounts of data, resulting in cost and time savings. In comparison to the traditional setting that requires "strong supervision" (i.e., a large number of high-quality labeled tuples are available to train the classifier), we collectively refer to these approaches as **classification with weak supervision**.

Other forms of weak supervision exist. To name a few, *crowdsourcing learning* aims to train a classification model with a *noisy training set*. Here, the class labels are provided by workers on a crowdsourcing platform (e.g., Amazon Mechanical Turk), where we can often obtain a large amount of labeled training tuples with a relatively low cost. However, some (or many) labels provided by the crowdsourcing workers might be wrong. How to infer the true label (i.e., the ground truth) from the noisy labels is a major concern of crowdsourcing learning. Crowdsourcing learning can be viewed as a form of weakly supervised learning in that the supervision (i.e., labels) is noisy or inaccurate. In *multi-instance learning*, each training tuple (e.g., an image, a document) is called a *bag*, which consists of a set of *instances* (e.g., different regions of an image, different sentences of a document). A bag is labeled as a positive bag, as long as at least one of its instances is assigned with a positive class label. A bag is labeled as a negative bag if none of its instances has a positive class label. For example, an image is labeled as "beach" if at least one of its regions is about beach; and it is labeled as "nonbeach" if none of its regions is about beach. Given a set of labeled bags, the goal of multi-instance learning is to train a classifier to predict the label of a test (previously unseen) bag. Multi-instance learning can be viewed as a form of weakly supervised learning, in that the label (i.e., supervision) is provided at a coarse granularity (i.e., at the bag level instead of instance level). The label of a bag is also called *group-level* label (e.g., a group of regions of an image, a group of sentences of a document).

7.5.1 Semisupervised classification

Semisupervised classification uses both labeled data and unlabeled data to build a classifier. Let $X_l = \{(x_1, y_1), \ldots, (x_l, y_l)\}$ be the set of labeled data and $X_u = \{x_{l+1}, \ldots, x_n\}$ be the set of unlabeled data. Here we describe a few examples of this approach for learning.

Self-training is the simplest form of semisupervised classification. It first builds a classifier using the labeled data. The classifier then tries to label the unlabeled data. The tuple with the most confident label prediction is added to the set of labeled data, and the process repeats (Fig. 7.16). Although the method is easy to understand, a disadvantage is that it may reinforce errors.

Cotraining is another form of semisupervised classification, where two or more classifiers teach each other. Each learner uses a different and ideally independent set of features for each tuple. Consider web page data, for example, where attributes relating to the images on the page may be used as one set of features, whereas attributes relating to the corresponding text constitute another set of features for the same data. Each set of features (called "a view") should be sufficient to train a good classifier. Suppose we split the feature set into two sets and train two classifiers, f_1 and f_2, where each classifier is trained on a different set. Then, f_1 and f_2 are used to predict the class labels for the unlabeled data, X_u. Each classifier then teaches the other in that the tuple having the most confident prediction from f_1 is added to the set of labeled data for f_2 (along with its predicted label).

Similarly, the tuple having the most confident prediction from f_2 is added to the set of labeled data for f_1. The method is summarized in Fig. 7.16. Cotraining is less sensitive to errors than self-training.

Self-training

1. Select a learning method such as Bayesian classification. Build the classifier using the labeled data, X_l.
2. Use the classifier to label the unlabeled data, X_u.
3. Select the tuple $x \in X_u$ having the highest confidence (most confident prediction). Add it and its predicted label to X_l.
4. Repeat (i.e., retrain the classifier using the augmented set of labeled data).

Cotraining

1. Define two separate nonoverlapping feature sets for the labeled data, X_l.
2. Train two classifiers, f_1 and f_2, on the labeled data, where f_1 is trained using one of the feature sets and f_2 is trained using the other.
3. Classify X_u with f_1 and f_2 separately.
4. Add the most confident $(x, f_1(x))$ to the set of labeled data used by f_2, where $x \in X_u$. Similarly, add the most confident $(x, f_2(x))$ to the set of labeled data used by f_1.
5. Repeat.

FIGURE 7.16

Self-training and cotraining methods of semisupervised classification.

A difficulty is that the assumptions for its usage may not hold true, that is, it may not be possible to split the features into mutually exclusive and class-conditionally independent sets.

Alternate approaches to semisupervised learning exist. For example, we can model the joint probability distribution of the features and the labels. For the unlabeled data, the labels can then be treated as missing data. The EM algorithm (Chapter 9) can be used to maximize the likelihood of the model. Semisupervised classification methods using support vector machines have also been proposed.

"*When does semisupervised classification work?*" Generally speaking, there are two commonly used assumptions behind semisupervised learning. The first assumption is *clustering assumption*, which means that data tuples from the same cluster are likely to share the same class label. The clustering algorithms will be introduced in Chapters 8 and 9. A representative example that utilizes the clustering assumption is semisupervised support vector machines (S3VMs). Recall that in the standard SVMs (Section 7.3), we seek a max-margin hyperplane that correctly separates the positive training tuples from negative tuples with a large margin. In S3VMs, it considers two design objectives, including (1) seeking a max-margin hyperplane to separate positive tuples from negative ones (which is the same as standard SVMs) and (2) avoiding to disrupt the clustering structure of unlabeled tuples. For the latter, this means that we favor a classifier (e.g., a hyperplane) that goes through the low-density region of the unlabeled tuples. The second commonly used assumption behind semisupervised learning is manifold assumption. We will not go into the technical details of manifold.[8] Simply put, the manifold assumption in the contexts of classification means that a pair of close tuples are likely to share the same class label. A representative example that utilizes the manifold assumption is graph-based semisupervised classification. It works as follows. First, we construct a graph whose nodes are input tuples, including both labeled and unlabeled tuples, and the edges indicate the local proximity. For example, we can link each data tuple to its k-nearest neighbors. In the constructed graph, only a small handful of nodes are

[8] In mathematical terms, a manifold is a topological space that approximates the Euclidean space in the vicinity of each data point.

labeled and the vast majority are unlabeled. The classification method propagates the labels of these labeled nodes (i.e., tuples) to the unlabeled nodes.

7.5.2 Active learning

Active learning is an iterative type of supervised learning that is suitable for situations where data are abundant, yet the class labels are scarce or expensive to obtain. The learning algorithm is active in that it can purposefully query a user (e.g., a human annotator) for labels. The number of tuples used to learn a concept this way is often much smaller than the number required in typical supervised learning.

"*How does active learning work to overcome the labeling bottleneck?*" To keep costs down, the active learner aims to achieve high accuracy using as few labeled instances as possible. Let D be all of data under consideration. Various strategies exist for active learning on D. Fig. 7.17 illustrates a *pool-based approach* to active learning. Suppose that a small subset of D is class-labeled. This set is denoted L. U is the set of unlabeled data in D. It is also referred to as a pool of unlabeled data. An active learner begins with L as the initial training set. It then uses a *querying function* to carefully select one or more data samples from U and requests labels for them from an oracle (e.g., a human annotator). The newly labeled samples are added to L, which the learner then uses in a standard supervised way. The process repeats. The goal of active learning is to achieve high accuracy using as few labeled tuples as possible. Active learning algorithms are typically evaluated with the use of learning curves, which plot accuracy as a function of the number of instances queried.

Most of the active learning research focuses on how to *choose* the data tuples to be queried. Several frameworks have been proposed. *Uncertainty sampling* is the most common strategy, where the active learner chooses to query the tuples that it is the least certain how to label. *Query-by-committee* is

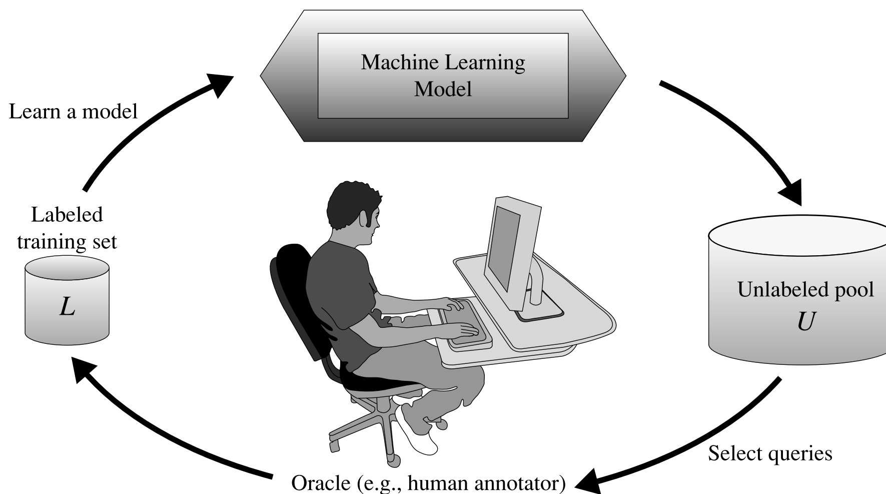


FIGURE 7.17

The pool-based active learning cycle. *Source:* From Settles [Set10], Burr Settles Computer Sciences Technical Report 1648, University of Wisconsin–Madison; used with permission.

another commonly used active learning strategy. In this method, it constructs multiple (say five) classification models and then selects the unlabeled tuple that constructed classification models have most disagreement in terms of its predicted class labels (say three classifiers predict that it belongs to positive class, whereas two classifiers predict it a negative tuple). Other strategies work to reduce the *version space*, that is, the subset of all hypotheses (i.e., classifiers) that are consistent with the observed training tuples. Alternatively, we may follow a decision-theoretic approach that estimates expected error reduction. This selects tuples that would result in the greatest reduction in the total number of incorrect predictions such as by reducing the expected entropy over U. This latter approach tends to be more computationally expensive.

7.5.3 Transfer learning

Suppose that an electronics store has collected a number of customer reviews on a product such as a brand of camera. The classification task is to automatically label the reviews as either positive or negative. This task is known as **sentiment classification**. We could examine each review and annotate it by adding a *positive* or *negative* class label. The labeled reviews can then be used to train and test a classifier to label future reviews of the product as either positive or negative. The manual effort involved in annotating the review data can be expensive and time consuming.

Now, suppose that the same store has customer reviews for other products as well, such as TVs. The distributions of review data for different types of products can vary greatly. We cannot assume that the TV-review data will have the same distribution as the camera-review data; thus we must build a separate classification model for the TV-review data. Examining and labeling the TV-review data to form a training set will require a lot of effort. In fact, we would need to label a large amount of data to train the review-classification models for each product. It would be nice if we could adapt an existing classification model (e.g., the one we built for cameras) to help learn a classification model for TVs. Such *knowledge transfer* would reduce the need to annotate a large amount of data, resulting in cost and time savings. This is the essence behind *transfer learning*.

Transfer learning aims to extract the knowledge from one or more *source tasks* and apply the knowledge to a *target task*. In our example, the source task is the classification of camera reviews, and the target task is the classification of TV reviews. Fig. 7.18 illustrates a comparison between traditional learning methods and transfer learning. Traditional learning methods build a new classifier for each new classification task, based on available class-labeled training and test data. Transfer learning algorithms apply knowledge about source tasks when building a classifier for a new (target) task. Construction of the resulting classifier requires fewer training data and less training time. Traditional learning algorithms assume that the training data and test data are drawn from the same distribution and the same feature space. Thus if the distribution changes, such methods need to rebuild the models from scratch.

Transfer learning allows the distributions, tasks, and even the data domains used in training and testing to be different. Transfer learning is analogous to the way humans may apply their knowledge of a task to facilitate the learning of another task. For example, if we know how to play the recorder, we may apply our knowledge of note reading and music to simplify the task of learning to play the piano. Similarly, knowing Spanish may make it easier to learn Italian.

Transfer learning is useful for common applications where the data becomes outdated or the distribution changes. Here we give two more examples. Consider *web-document classification*, where we may have trained a classifier to label, say, articles from various newsgroups according to predefined

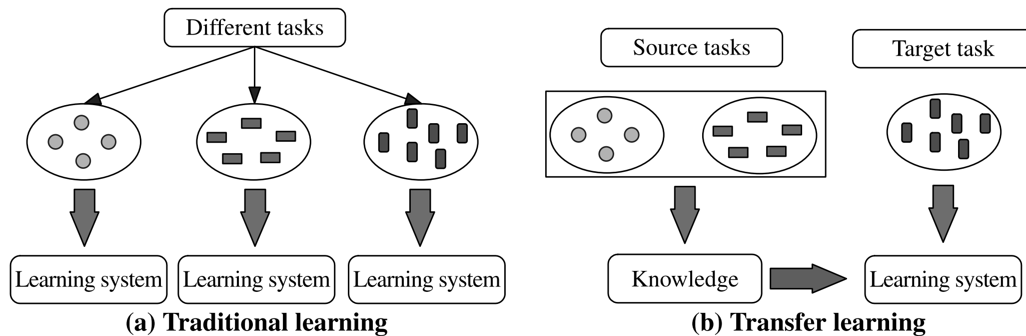


FIGURE 7.18

Transfer learning vs. traditional learning. (a) Traditional learning methods build a new classifier from scratch for each classification task. (b) Transfer learning applies knowledge from a source classification task to simplify the construction of a classifier for a new, target classification task. *Source:* From Pan and Yang [PY10]; used with permission.

categories. The web data that were used to train the classifier can easily become outdated because the topics on the Web change frequently. Another application area for transfer learning is *email spam filtering*. We could train a classifier to label email as either "*spam*" or "*not spam*," using email from a group of users. If new users come along, the distribution of their email can be different from the original group, hence the need to adapt the learned model to incorporate the new data.

There are various approaches to transfer learning, the most common of which is the *instance-based transfer learning* approach. This approach reweights some of the data from the source task and uses it to learn the target task. The **TrAdaBoost** (Transfer AdaBoost) algorithm exemplifies this approach. Consider our previous example of web-document classification, where the distribution of the old data on which the classifier was trained (the source data) is different from the newer data (the target data). TrAdaBoost assumes that the source and target domain data are each described by the same set of attributes (i.e., they have the same "feature space") and the same set of class labels, but that the distributions of the data in the two domains are very different. It extends the AdaBoost ensemble method described in Section 6.7.3. TrAdaBoost requires the labeling of only a small amount of the target data. Rather than throwing out all the old source data, TrAdaBoost assumes that a large amount of it can be useful in training the new classification model. The idea is to filter out the influence of any old data that are very different from the new data by automatically adjusting weights assigned to the training tuples.

Recall that in boosting, an ensemble is created by learning a series of classifiers. To begin, each tuple is assigned a weight. After a classifier M_i is learned, the weights are updated to allow the subsequent classifier, M_{i+1}, to "pay more attention" to the training tuples that were misclassified by M_i. TrAdaBoost follows this strategy for the target data. However, if a source data tuple is misclassified, TrAdaBoost reasons that the tuple is probably very different from the target data. It therefore *reduces* the weight of such tuples so that they will have less effect on the subsequent classifier. As a result, TrAdaBoost can learn an accurate classification model using only a small amount of new data and a large amount of old data, even when the new data alone are insufficient to train the model. Hence in this way TrAdaBoost allows knowledge to be transferred from the old classifier to the new one.

A major challenge with transfer learning is **negative transfer**, which occurs when the new classifier performs worse than if there had been no transfer at all. Work on how to avoid negative transfer is an area of active research, where the key is to quantify the difference between the source task and the target task. *Heterogeneous transfer learning*, which involves transferring knowledge from different feature spaces and multiple source domains, is another active research topic. Traditionally, transfer learning has been used on small-scale applications. The use of transfer learning on larger applications, such as social network analysis and video classification, is often built upon the deep learning models with a "pretraining" plus "fine-tuning" strategy, which will be introduced in Chapter 10.

Transfer learning is closely related to another powerful weakly supervised learning method, namely *multitask learning*.[9] Let us use the sentiment classification example to illustrate the difference between transfer learning and multitask learning. In the transfer learning setting, we assume that we have a large number of manually labeled camera review data (i.e., the source task), but a very limited number of manually labeled TV review data (i.e., the target task). The goal of transfer learning to transfer the knowledge about the source task (camera review sentiment classification) to help build a better classifier for TV review sentiment classification (i.e., the target task). Now, suppose for both TV review and camera review, we only have a small amount of manually labeled data. How can we accurately build both classifiers—one for TV review sentiment and the other for camera review sentiment? Multitask learning addresses this challenge by training both classifiers simultaneously so that the knowledge from one learning task (e.g., TV review sentiment) can be transferred to the other learning task (e.g., camera review sentiment), and vice versa.

7.5.4 Distant supervision

Let us take another look at the sentiment classification example. Suppose that an electronics store launches a new holiday sales campaign on social media platforms (e.g., Twitter), which goes viral with hundreds of thousands tweets. The store manager wants to figure out the sentiment of these Tweets, so that she can adjust the campaign strategy accordingly. We could manually label a large number of tweets regarding their sentiment and then train a classifier to predict the sentiment (positive vs. negative) of the remaining tweets. However, that would be time consuming. The manager wonders: "*Can we train a sentiment classifier about the tweets without any manual labels?*" **Distant supervision** aims to answer this question by automatically generate a large number of labeled tuples. In particular, the manager notices that for a large subset of the tweets, its text content contains a ":)" sign or a ":(" sign, which are often associated with positive and negative sentiments, respectively. Therefore we could treat all the tweets with a ":)" sign as positive tuples and those with a ':(' sign negative tuples and use them to train a sentiment classifier. Once the classifier is trained, we can use it to predict the sentiment for any future tweet even if it does not contain a ":)" or ":(" sign. Notice that in this case, we do not need to manually label *any* tweet in terms of its sentiment, and such labels (regarding positive or negative sentiment) are automatically generated.

In the tweet sentiment classification example above, we exploit the specific information (i.e., a ":)" or ":(" sign) in the input data to automatically generate labeled training tuples. An alternative strategy for classification with distant supervision often leverages the external knowledge base to automatically

[9] In some machine learning literature, multitask learning is viewed as a special case of transfer learning, namely inductive transfer learning where the source and target domains share the same feature (i.e., attribute) space.

generate labels for the training tuples. For example, in order to classify tweets into different categories (e.g., news, health, science, games, etc.), we could explore the Open Directory Project (ODP, http://odp.org), which maintain a directory for web links by volunteers. Thus if a tweet contains a url (e.g., http://nytimes.com), we can automatically find its ODP category (e.g., news), which is treated as the label of the corresponding tweet. In this way, we will be able to automatically generate a large labeled training set. Once the classifier is trained, we can use it to predict the class label (i.e., the category) of a test tweet, even if it does not contain a url. Another way to automatically generate labeled training examples is to leverage YouTube video that is linked to the tweet. The method is based on the following two observations. First, there are a large number of tweets, each of which contains a link to a YouTube video. Second, for each YouTube video, it is always associated with one of 18 predefined class labels. Therefore we can treat the label of YouTube video as the label of the associated tweet.

In addition to social media post classification tasks, distant supervision is also found useful for relation extraction for natural language processing. An active research direction in distant supervision is how to effectively ask users to write a *labeling function*, instead of manually label training tuples, to automatically generate labels for a large number of unlabeled data. A major limitation of distant supervision is that the automatically generated labels are often very noisy. For example, some tweets with a ":)" sign could have neutral or even negative sentiment; the class labels of a tweet does not always align with the label or category of the url (either a web page or a YouTube video) it contains.

7.5.5 Zero-shot learning

Suppose that we have a collection of animal images, each of which has a unique label, including "owl," "dog," or "fish." Using this training data set, we can build a classifier, say SVMs or logistic regression classifier.[10] Then, given a test image, we can use the trained classifier to predict its class label, that is, which one of the three possible animals (owl, dog, or fish) this image is about. *But, what if the test image is actually about a cat?* In other words, the class label of the test data *never* appears in the training data. This is what *zero-shot learning* aims to address, where the classifier needs to predict a test tuple whose class label was never observed during the training stage. In other words, there is *zero* training tuples for the novel class label (e.g., cat in our example). The term "shot" here refers to data tuple.

At the first glance, this seems to be an impossible mission. You might wonder: "*If there is zero training tuples about the cat, how can I build a classifier to recognize an image about the cat?*" However, we might have some high-level description about the novel classes. For example, for "cat," we can learn from the Wikipedia that a cat has retractable claws and super night vision. Zero-shot learning tries to leverage such external knowledge or side-information to build a classifier that can recognize such novel class labels.

Let us use the animal classification example (Fig. 7.19) to explain how zero-shot learning works. Formally, there are n training images each of which is represented by a d-D feature vector and a 3-D label vector. The label vector indicates which of the three known classes the training image belongs to. For example, for an image about a "dog," its label vector is $[1, 0, 0]$. In addition, we have the external

[10] Different from the classification tasks we have seen so far which typically involve two possible class labels (e.g., positive vs. negative sentiment), in this setting, we have a multiclass classification problem since there are three possible class labels. The techniques for multiclass classification will be introduced in Section 7.7.1.

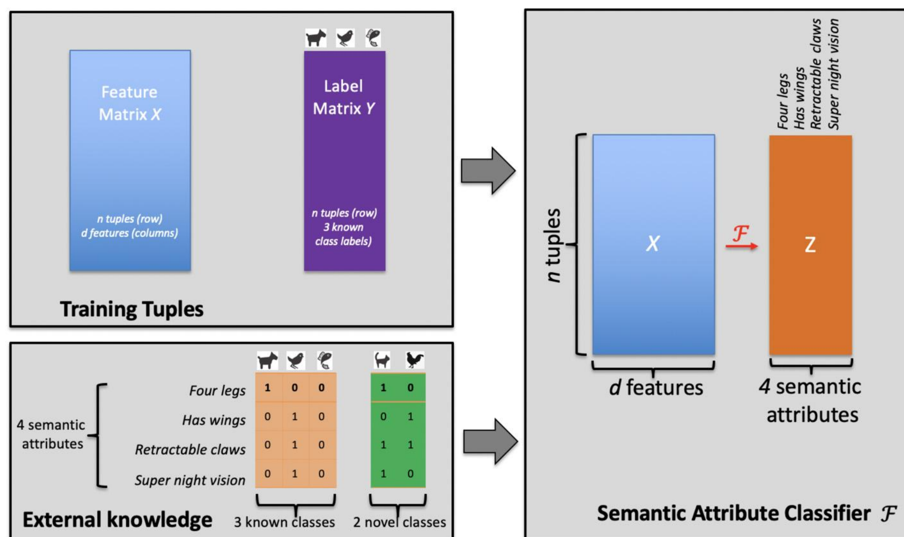


FIGURE 7.19

Top left: input n training tuples in d-dimensional feature space, each of which is labeled by one of the three known classes (i.e., "dog," "owl," and "fish"). Bottom left: external knowledge where each known and novel class is described by four semantic attributes. Right: the trained semantic attribute classifier $\mathcal{F}$.

knowledge about the class label, where each class label (animal) can be described by four *semantic attributes*,[11] including whether the animal "has four legs," "has wings," "has retractable claws," and "has super night vision." For example, since a dog has four legs, but no wings or retractable claws or super night vision, the class label "dog" can be described by a 4-D semantic attribute vector $[1, 0, 0, 0]$. Likewise, the class label "cat" can be described by a 4-D semantic attribute vector $[1, 0, 1, 1]$, meaning that a cat has four legs, retractable claws and super night vision but no wings. Notice that such external knowledge is available for both known class labels (e.g., "owl," "dog," and "fish") and novel class labels (e.g., "cat" and "rooster").

Then, using the input training tuples (i.e., the $n \times d$ feature matrix X and the $n \times 3$ label matrix Y in the upper left corner of Fig. 7.19) and the external knowledge about the three known class labels (i.e., the information about four semantic attributes for the three known class labels in the bottom left corner of Fig. 7.19), we train a *semantic attribute classifier* $\mathcal{F}$, which predicts a 4-D semantic attribute vector for an input image represented by a d-dimensional feature vector. In our example, the output of the semantic attribute classifier $\mathcal{F}$ tells whether the given image has "four legs," "wings," "retractable claws" and "super night vision," respectively. We can use a two-layer neural network to train such a

[11] In the literature, the semantic attribute is also referred to as semantic feature or semantic property or just attribute.

semantic attribute classifier, which will be introduced in Chapter 10.[12] Then, given a test image, we predict which of the two novel classes (i.e., "cat" and "rooster") it belongs to based on the following two steps. First, given the d-D feature vector of the test image, we use the semantic attribute classifier $\mathcal{F}$ to output a 4-D semantic attribute vector, whose elements indicate whether or not the test image has the corresponding semantic attributes. For example, if the semantic attribute classifier output a vector $[1, 0, 0, 1]$, it means that the classifier predicts that the test image (1) has four legs, (2) has no wings, (3) has no retractable claws, and (4) has super night vision. Second, we compare the predicted semantic attribute vector with the external knowledge about the two novel classes, respectively (i.e., the 4×2 green (dark gray in print version) table in the middle bottom of Fig. 7.19). We predict that the test image belongs to the novel class whose semantic attribute vector is most similar to that of the test image. In our example, since the predicted semantic attribute vector $[1, 0, 0, 1]$ is more similar to that of "cat" ($[1, 0, 1, 1]$) than that of "rooster" ($[0, 1, 1, 0]$), we predict that it is an image about "cat."

The key of the method described above is that we leverage the semantic attributes as a bridge to transfer the output of the semantic classifier that was trained on the known class labels to predict the novel class labels. From this perspective, we can also view zero-shot learning as a special form of transfer learning (i.e., to transfer the knowledge about the known class labels to novel classes). In addition to the semantic attribute, there are other forms of external knowledge that can be harnessed for zero-shot learning. An example is the class-class similarity between known and novel classes. In the animal image classification application mentioned above, we can train a multiclass classifier to predict which of the three known classes an image belongs to. Now, given a test image that comes from the novel class (either "cat" or "rooster"), the trained classifier predicts it belongs to "dog," and if we know that "dog" is more similar to "cat" than "rooster," it is safe to predict the test image is indeed a "cat," rather than a "rooster." In the standard zero-shot learning setting, we always assume that the test image must come from one of the novel classes. This assumption might be too strong in reality. For example, the test image might come from either known classes (dog, owl, or fish) or novel classes (cat or rooster). There have been research on *generalized zero-shot learning* to address such a more complicated setting. Other applications of zero-shot learning include neural activity recognition, where the classifier needs to recognize the word that a person is thinking about based her neural activity reflected on the fMRI image. In this application, the class labels are words. It is impossible to construct a training data set that covers all possible words that a human can think of. Zero-shot learning can effectively help extrapolate the classifier trained on a limited number of words (known class labels) to the unseen words during the training stage (i.e., the novel classes).

7.6 Classification with rich data type

The classification techniques we have seen so far assume the following setting. That is, given a training set, where each training tuple is represented by a feature (or attribute) vector and a class label, we build

[12] The input of this two-layer neural network is the d-D feature, the hidden layer corresponds to the four semantic attributes, and output layer corresponds to the three known class labels. Unlike a typical neural network, the model parameter for the second layer (from semantic attribute to the known class labels) can be directly obtained based on the external knowledge about four semantic attributes for the three known class labels.